НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

«На правах рукопису»
УДК _004.94, 004.42_

До захисту допущено:
Завідувач кафедри
_____ О. І. Ролік
«___»_____2021 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Програмна інженерія»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації»

Виконав (-ла):
студент (-ка) VI курсу, групи ІТ-91мн
Сопов Олексій Олександрович _____

Керівник:
декан ФІОТ, д.т.н., професор
Теленик Сергій Федорович _____

Консультант:
доцент кафедри, к.т.н. доцент
Жаріков Едуард В'ячеславович _____

Рецензент:
проф. кафедри ММСА ІПСА, д.т.н. професор
Бідюк Петро Іванович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – другий (магістерський)
Спеціальність – 121 «Інженерія програмного забезпечення»
Освітньо-наукова програма «Програмна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____О. І. Ролік
«___»_____20__ р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Сопову Олексію Олександровичу

1. Тема дисертації « **Планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації** », науковий керівник дисертації професор, д.т.н. Теленик С. Ф., затверджені наказом по університету від «12» березня 2021 р. №_809_

2. Термін подання студентом дисертації: 11.05.2021

3. Об'єкт дослідження: системи оркестрації контейнерами та алгоритми планування ресурсної ємності.
4. Предмет дослідження: методи та алгоритми планування ресурсної ємності для заданого контейнерного навантаження, інструментальні засоби для планування ресурсної ємності.

5. Перелік завдань, які потрібно розробити: аналіз предметної області контейнерної віртуалізації та контейнерних оркестраторів; розробка математичної моделі проблеми; побудова алгоритму планування ресурсної ємності; проведення експериментів; розробка інструментальних засобів для планування ресурсної ємності

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: Діаграма компонентів Kubernetes; Діаграма активності планування ресурсної ємності; Діаграма активності додавання нового пода до кластера; Діаграма розгортання системи; Діаграма прецедентів системи; Діаграма компонентів системи; Діаграма класів застосунку; Діаграма послідовності обробки запитів.

7. Орієнтовний перелік публікацій: Особливості масштабування контейнерного навантаження на базі системи Kubernetes; Аналіз доступності мікросервісів на базі системи управління та оркестрації контейнерів Kubernetes; Покращення алгоритму розподілу навантаження між віртуальними машинами у Kubernetes.

8. Дата видачі завдання 01.02.2021

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Отримання та узгодження теми магістерської дисертації	10.01.2021 – 01.02.2021	
2	Аналіз теоретичних матеріалів та вивчення предметної області	02.03.2021 – 18.02.2021	
3	Розробка математичної моделі	19.02.2021 – 05.03.2021	
4	Моделювання досліджених та створених алгоритмів. Проведення експериментів	06.03.2021 – 25.03.2021	
5	Розробка програмної системи	26.03.2021 – 15.04.2021	
6	Оформлення текстових та графічних матеріалів	16.04.2021 – 04.05.2021	
7	Передзахист магістерської дисертації	05.05.2021	
8	Доопрацювання пояснювальної записки та підготовка презентаційного матеріалу	06.05.2021 – 17.05.2021	
9	Подача магістерської дисертації	11.05.2021	
10	Захист магістерської дисертації	18.05.2021	

Студент Олексій СОПОВ

Науковий керівник Сергій ТЕЛЕНИК

РЕФЕРАТ

Сопов О. О. Планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації: 140 сторінок, 34 рисунків, 6 таблиць та 10 додатків.

Ключові слова: віртуалізація, контейнер, контейнерний оркестратор, Docker, Kubernetes, алгоритм, ресурсна ємність, математична модель.

Магістерська дисертація присвячена розробці алгоритму планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації.

В роботі проаналізовано особливості контейнерної віртуалізації, досліджено системи оркестрації контейнерами та їх особливості, проаналізовано проблему планування ресурсної ємності. Для роботи із контейнерами найчастіше використовують контейнерні оркестратори, які використовують примітивні алгоритми розподілення навантаження.

Побудована математична модель проблеми. У даній роботі задача планування класифікується як задача багатовимірного пакування та задача багатовимірного пакування із контейнерами різного розміру.

Розроблено алгоритм планування, який базується на відомих евристичних алгоритмах, та запропоновано власну евристику, яка показує кращі результати. Розроблено власні методи переходу між багатовимірним вектором ресурсів у скалярне значення. Проведено експерименти, які підтверджують високу утилізацію ресурсів та результат алгоритму близький до оптимального.

Розроблено інструментарні засоби для вирішення поставленої задачі.

ABSTRACT

Sopov O. Resource capacity planning for a given load in terms of container virtualization: 140 pages, 35 figures, 6 tables and 10 appendices.

Keywords: virtualization, container, container orchestrator, Docker, Kubernetes, algorithm, resource capacity, mathematical model.

The master's dissertation is devoted to development of algorithm of planning of resource capacity for the set loading in the conditions of container virtualization.

The features of container virtualization are analyzed in the work, the systems of orchestration by containers and their features are investigated, the problem of resource capacity planning is analyzed. To work with containers, container orchestrators are most often used, which use primitive load distribution algorithms.

The problem of resource capacity planning is analyzed, a mathematical model is built. In this paper, the planning task is classified as a multidimensional packaging problem and a multidimensional packaging problem with containers of different sizes.

A scheduling algorithm based on known heuristic algorithms has been developed, and our own heuristics are proposed, which show the best results. Own methods of transition between multidimensional resource vector to scalar value are developed. Experiments were performed, which confirm the high utilization of resources and the result of the algorithm is close to optimal.

Developed tools for solving the problem.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CPU – Central processing unit: центральний процесов

CFS – Completely Fair Scheduler: повністю чесний планувальник

ОС – Операційна система

Оркестрація – автоматизована координація, конфігурація, та управління системою та програмним забезпеченням

Под – Абстракція у Kubernetes, що описує один або декілька контейнерів

AWS – Amazon Web Services

API – Application programming interface: прикладний програмний інтерфейс

ЦОД – Центр опрацювання даних

ВСТУП

Технологія віртуалізації в останній час поступово замінює класичний підхід, коли програмне забезпечення встановлюється безпосередньо на фізичні окремі фізичні машини, сервери. Використання віртуалізації дозволяє покращити рівень утилізації ресурсів, підвищити швидкість створення нового програмного оточення, та, безпосередньо, зменшити ціну інфраструктури через те, що на одній фізичній машині можливо запустити велику кількість програмних застосунків.

Під віртуалізацією мається на увазі набір технологій, які надають можливість створення набору обчислювальних ресурсів або їх логічного об'єднання, абстраговані від апаратної реалізації із забезпеченням при цьому логічної ізоляції один від одного обчислювальних процесів, які виконуються на одному фізичному ресурсі. Яскравим прикладом віртуалізації може бути віртуальна машина.

Все більшого розвитку набуває технологія контейнерної віртуалізації, або ж, віртуалізації на рівні операційної системи. При такому методі віртуалізації ядро операційної системи підтримує декілька ізольованих просторів користувача, замість одного. Ядро забезпечує повну ізольованість контейнерів, тому програми із різних контейнерів не можуть впливати одне на одного. Використання такого типу віртуалізації знижує накладні витрати, що дозволяє використовувати надані ресурси більш ефективно.

В реальному світі перед розробниками дуже часто постає задача у підтримці та моніторингу великої кількості контейнерів. Для вирішення цієї задачі було створено велику кількість систем оркестрації контейнерів. Проте, незважаючи на те, що такі системи покривають велику кількість питань у сфері керування контейнерами, вони не надають механізму планування ресурсної ємності кластерів, на яких контейнери буде запущено. Також, зазвичай, використовуються не найбільш оптимальні алгоритми розміщення контейнерів у кластері.

Метою роботи є підвищення ефективності використання ресурсів кластера віртуальних машин за допомогою алгоритму планування ресурсної ємності для

заданого навантаження в умовах контейнерної віртуалізації для системи оркестрації контейнерами. Для цього необхідно вирішити наступні завдання:

- аналіз особливостей контейнерної віртуалізації.
- аналіз систем оркестрації контейнерами, дослідження їх особливостей;
- аналіз проблеми планування ресурсної ємності;
- розробка математичної моделі проблеми, аналіз існуючих алгоритмів;
- побудова алгоритму планування ресурсної ємності, проведення експериментальних досліджень із використанням розробленого алгоритму;
- розробка інструментальних засобів для планування ресурсної ємності та програми для формування кластера ресурсів для керування заданим навантаженням на базі однієї з систем оркестрації.

Об’єктом дослідження даної роботи є системи оркестрації контейнерами та алгоритми планування ресурсної ємності.

Предметом дослідження є методи та алгоритми планування ресурсної ємності для заданого контейнерного навантаження; інструментальні засоби для планування ресурсної ємності.

У даній роботі використано такі методи дослідження:

- метод математичного моделювання – для розробки математичної моделі проблеми, побудови алгоритму планування;
- метод програмного моделювання – для проведення експериментів, створення інструментальних засобів для планування ресурсної ємності.

Наукова новизна одержаних результатів полягає у вирішенні науково-практичного завдання розроблення алгоритму планування ресурсної ємності кластера в умовах контейнерної віртуалізації, у дослідженні алгоритмів багатовимірного пакування для задачі розподілення контейнерів у кластері ресурсів, створенні власної евристики для задачі двовимірного пакування.

Одержані в даній роботі результати можуть бути використані приватними підприємствами для правильного планування ресурсної ємності кластера ресурсів, при плануванні бюджету на розробку. Також, одержані результати можуть бути використані розробниками систем оркестрації контейнерами для надання

кінцевому користувачу більш широкої інформації із питань формування кластера ресурсів, для покращення алгоритмів розподілу контейнерів.

Результати роботи опубліковані у журналі «Технічні науки та технології» №1(23), 2021, м. Чернігів у статті «Особливості масштабування контейнерного навантаження на базі системи Kubernetes», у журналі «Проблеми інформатизації та управління» , Том 1, №65, 2021, м. Київ у статті «Аналіз доступності мікросервісів на базі системи управління та оркестрації контейнерів Kubernetes» та у журналі восьмої науково практичної конференції «Science and education: problems, prospects and innovations» у статті «Покращення алгоритму розподілу навантаження між віртуальними машинами у кластері Kubernetes»

1 ВІРТУАЛІЗАЦІЯ ЯК СПОСІБ ОПИСУ ЗАСТОНУНКУ ТА ЙОГО РЕСУРСІВ

Для правильного планування ресурсної ємності котнейнерного навантаження, або ж віртуалізації на рівні операційної системи, необхідно розуміти основні концепти роботи віртуалізації, її переваги та недоліки. У даному розділі описано поняття віртуалізації, порівняно два найпопулярніших типи віртуалізації: віртуальна машина та контейнер, показано зв'язок контейнера із розроблюваним застосунком та його ресурсами.

1.1 Поняття віртуалізації

Віртуалізація – це технологія, яка дозволяє надавати та створювати корисні IT-послуги, використовуючи ресурси, традиційно пов'язані з апаратним забезпеченням. Використання віртуалізації дозволяє використовувати повну потужність фізичної машини, розподіляючи її ресурси серед багатьох користувачів чи середовищ [1].

Наприклад є 3 фізичні сервери з окремими спеціальними цілями. Один – це сервіс призначений для пошти, інший – веб-сервер із програмним застосунком, а останній запускає внутрішні застарілі програми. Кожен сервер використовується приблизно на 30% ємності, що складає невелику частку їх робочого потенціалу. Але оскільки застарілі програми залишаються важливими для внутрішніх операцій, то необхідно запускати і третій сервер. Приклад такого використання ресурсів зображено на рисунку 1.

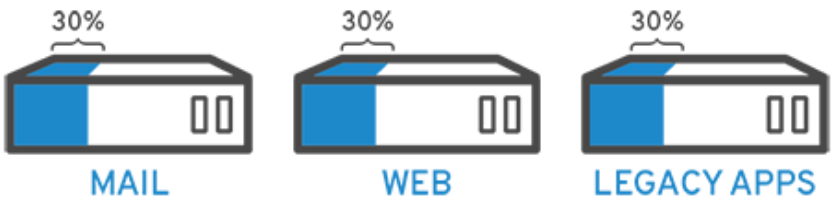


Рисунок 1.1 – Приклад використання ресурсів без використання технологій віртуалізації

Часто було простіше і надійніше запускати окремі завдання на окремих серверах: 1 сервер, 1 операційна система, 1 завдання. Можливість помістити декілька незалежних зон на одному фізичному сервері було доволі складно. Але із використанням технологій віртуалізації можливо розділити поштовий сервер на 2 унікальних, які можуть обробляти незалежні завдання, щоб застарілі програми можна було перенести. Таким чином, один фізичний сервер виконує вже дві задачі, проте має достатньо ресурсів і для виконання інших. Приклад розділення одного серверу на декілька віртуальних із використанням технологій віртуалізації зображено на рисунку 2.

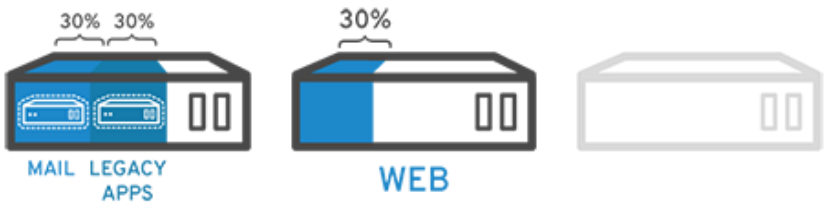


Рисунок 1.2 – Приклад використання технологій віртуалізації для першого сервера

Беручи до уваги безпеку, перший сервер можна розділити так, щоб він вмістив і більшу кількість завдань – збільшивши утилізацію ресурсів з 30%, до 60%, до 90%. Після цього порожні сервери можуть бути використані для інших завдань або взагалі припинені, щоб зменшити витрати на охолодження та обслуговування.

Хоча технологія віртуалізації була розроблена ще в 1960-х роках, вона не була широко застосована до початку 2000-х [2]. Технології, що уможливили віртуалізацію, такі як гіпервізори, були розроблені десятки років тому, щоб надати кільком користувачам одночасний доступ до комп'ютерів, які виконували пакетну обробку. Раніше пакетна обробка була популярним підходом у бізнісі, для швидкого виконання рутинних задач.

Але протягом наступних кількох десятиліть популярність інших рішень для багатьох користувачів / окремих машин зростала, а віртуалізація – ні. Одним з цих

інших рішень був розподіл часу, який ізолював користувачів в операційних система – ненавмисно приводячи до інших операційних систем, таких як UNIX, які з часом поступилися місцем Linux. Протягом усього часу віртуалізація залишалася в основному неприйнятою нішевою технологією.

Швидко вперед до 1990-х. Більшість підприємств мали фізичні сервери та IT-стеки одного постачальника, що не дозволяло застарілим програмам працювати на іншому обладнанні постачальника. Оскільки компанії оновлювали свої IT-середовища менш дорогими товарними серверами, операційними системами та програмами різних постачальників, вони були змушені використовувати недостатньо фізичного обладнання – кожен сервер міг виконувати лише одне завдання, специфічне для постачальника.

На даному етапі віртуалізація стала використовуватися все ширше. Це було одночасним рішенням двох основних проблем: компанії могли розділити свої ресурси та запускати застарілі програми на декількох типах та версіях операційних систем. Сервери почали використовуватись більш ефективно (або взагалі не використовуватись), зменшуючи тим самим витрати, пов’язані з придбанням, налаштуванням, охолодженням та обслуговуванням.

Широке застосування віртуалізації допомогло зменшити блокування постачальників і зробило це основою хмарних обчислень. Сьогодні воно настільки поширене на підприємствах, що для відстеження всього цього часто потрібно спеціалізоване програмне забезпечення для управління віртуалізацією.

Якщо віртуалізація визначається як можливість роботи декількох операційних систем на одному хост-комп'ютері, то найважливішим компонентом стеку віртуалізації є гіпервізор. Цей гіпервізор, який також називають монітором віртуальних машин, створює віртуальну платформу на головному комп'ютері, поверх якої виконується та контролюється кілька гостьових операційних систем. Таким чином, кілька операційних систем, які є або декількома екземплярами однієї і тієї ж операційної системи, або різних операційних систем, можуть спільно використовувати апаратні ресурси, пропоновані хостом. Приклад роботи гіпервізора показано на рисунку 1.3.

Існує два основних типи гіпервізорів:

– нативні, або апаратні – це системи, які працюють безпосередньо на апаратному забезпеченні хоста для управління апаратним забезпеченням та моніторингу гостьових операційних систем. Отже, гостьова операційна система працює на окремому рівні над гіпервізором. Прикладами цієї класичної реалізації архітектури віртуальних машин є Oracle VM, Microsoft Hyper-V, VMWare ESX та Xen.

– програмні – призначені для роботи в рамках традиційної операційної системи. Іншими словами, розміщений гіпервізор додає чіткий програмний рівень поверх хост-операційної системи, а гостьова операційна система стає третім програмним рівнем над апаратним забезпеченням. Відомим прикладом програмного гіпервізора є Oracle VM VirtualBox [3].

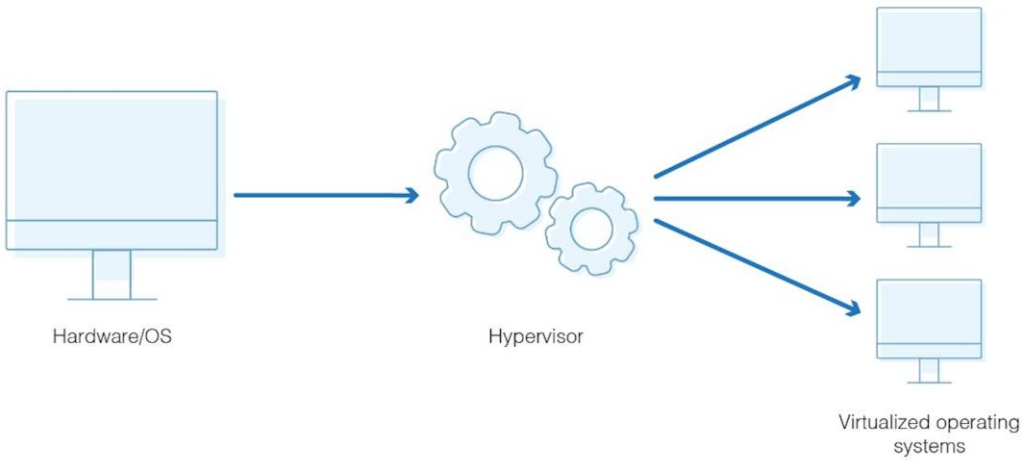


Рисунок 1.4 – Приклад роботи гіпервізора

Основними перевагами використання віртуалізації є:

- скорочення витрат на підтримку інфраструкутури – віртуалізація допомагає мінімізувати загальні витрати на підтримку інфраструктури для бізнесу. Невіртуальне середовище часто має низьку утилізацію ресурсів, що робить їх неефективними. Віртуалізоване середовище дозволяє максимізувати використання поточного обладнання, а не витрачати гроші на придбання

нової інфраструктури. Маючи менше обладнання, зокрема серверів, це також означає економію на обслуговуванні, платі за оренду та енергетичні витрати, пов’язані з підтримкою серверів. Більшість постачальників хмарних рішень пропонують послуги віртуалізації на підписках, що дозволяє платити лише за те, що насправді використовується [4]. Впровадження віртуального середовища також допомагає ефективніше формувати бюджети та розподіляти фінансові ресурси на інші сфери бізнесу;

– полегшання підтримки – заміна фізичних комп’ютерів програмно визначеними віртуальними машинами спрощує використання та управління політиками, написаними в програмному забезпеченні. Це дозволяє створювати автоматизовані робочі процеси з управління IT-послугами. Наприклад, засоби автоматизованого розгортання та конфігурації дозволяють адміністраторам визначати колекції віртуальних машин та додатків як послуги в шаблонах програмного забезпечення. Це означає, що вони можуть встановлювати ці послуги неодноразово та послідовно, не вимагаючи громіздких витрат часу, та схильного до помилок річного налаштування. Адміністратори можуть використовувати політики безпеки віртуалізації для призначення певних конфігурацій безпеки на основі ролі віртуальної машини. Політика може навіть підвищити ефективність використання ресурсів, вилучивши невикористані віртуальні машини для економії місця та обчислювальних потужностей;
– підвищення надійності – збої ОС та додатків можуть спричинити простої та порушити продуктивність користувачів. Адміністратори можуть запускати декілька надлишкових віртуальних машин поруч один з одним і переходити до відмови між ними, коли виникають проблеми. Запуск кількох надлишкових фізичних серверів є дорожчим.
– пришвидшення процесу розгортання програмних продуктів – купівля, встановлення та налаштування обладнання для кожної програми займає багато часу. За умови, що апаратне забезпечення вже встановлене, забезпечення віртуальних машин для запуску всіх ваших програм значно

швидше. Це можливо автоматизувати за допомогою програмного забезпечення для управління та вбудувати в існуючі процеси розробки.

У процесі розробки та розгортання програмного забезпечення найчастіше використовуються віртуальні машини. Віртуальна машина – це імітований еквівалент комп’ютерної системи, яка працює поверх іншої системи. Віртуальні машини можуть мати доступ до будь-якої кількості ресурсів: обчислювальної потужності за допомогою апаратного забезпечення, але обмеженого доступу до центрального процесора та пам'яті хост-машини; один або кілька фізичних або віртуальних дискових пристроїв для зберігання; віртуальний або реальний мережевий інтерфейс; а також будь-які пристрої, такі як відеокарти, USB-пристрої чи інше обладнання, що надається спільно з віртуальною машиною. Якщо віртуальна машина зберігається на віртуальному диску, це часто називають образом диска.

1.2 Контейнерна віртуалізація

Віртуалізація контейнерів (яку часто називають віртуалізацією операційної системи) – це більше, ніж просто інший тип гіпервізора. Контейнери використовують в якості основи операційну систему хоста, а не гіпервізор.

Замість того, щоб віртуалізувати апаратне забезпечення (що вимагає повноцінних віртуалізованих образів операційної системи для кожного гостя), контейнери віртуалізують саму ОС, обмінюючись з ядром ОС та її ресурсами як з хостом, так і з іншими контейнерами.

Контейнери надають найнеобхідніше, що потрібно для роботи будь-якої програми на головній ОС. Можна думати про про як про розірвані віртуальні машини, на яких працює лише достатньо програмного забезпечення для розгортання програми.

Наприклад, сервери баз даних, які найкраще працюють із використанням блочного сховища – вимагають прямого доступу до апаратних ресурсів. Отримання такого доступу до дисків та мережевих пристроїв через апаратну емуляцію

гіпервізора часто негативно впливає на їх продуктивність. Віртуалізація контейнерів допомагає, просто минаючи рівень емуляції. Контейнери використовують одне і те ж серверне обладнання та операційну систему, але над цим немає гіпервізора.

Технічно, саме ядро Linux не здогадується, що наверху живуть контейнери, але тим не менше контейнери можуть спільно використовувати ресурси ядра за допомогою таких функцій, як простори імен, cgroups та chroot. Вони дозволяють контейнерам ізолювати процеси, повністю управляти ресурсами та забезпечувати належну безпеку. Приклад використання контейнерної віртуалізації наведено на рисунку 1.5.

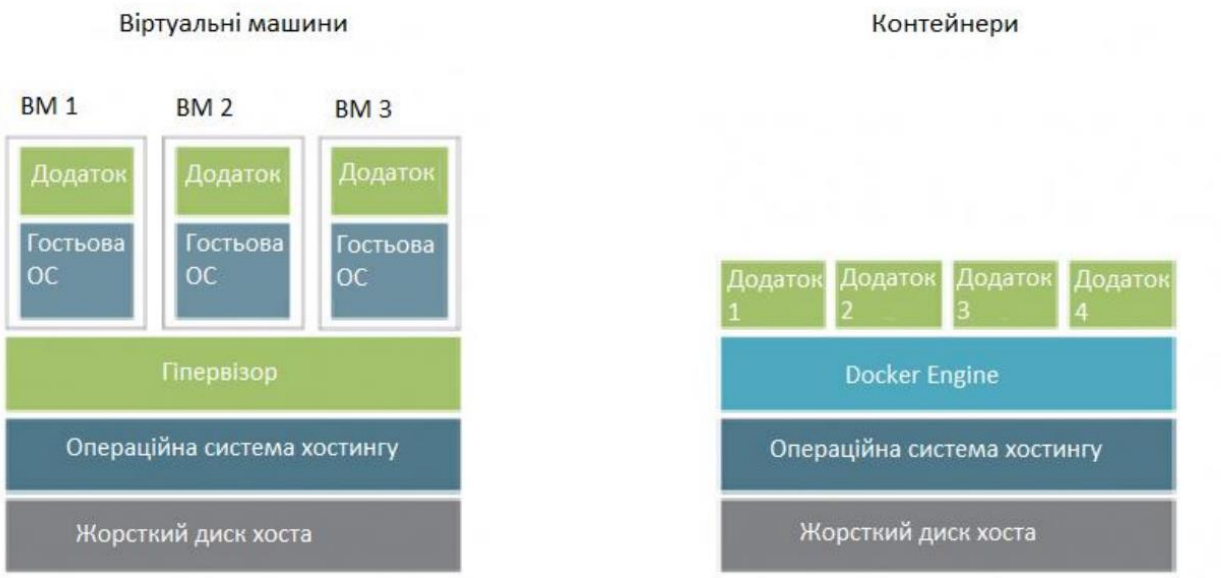


Рисунок 1.5 – Приклад використання контейнерної віртуалізації

Контейнери, як і віртуальні машини мають власне обмеження за ресурсами, та не впливають на продуктивність інших контейнерів. Контейнерна віртуалізація стала популярною через ряд переваг:

– зменшення ціни на апаратне забезпечення – віртуалізація за допомогою контейнерів зменшує апаратні витрати, дозволяючи консолідацію (тобто виділення декількох додатків до одного обладнання покращує використання обладнання). Це дозволяє одночасно виконуваному програмному

забезпеченню скористатися справжньою паралельністю, що забезпечується багатоядерною апаратною архітектурою. Крім того, це дозволяє системним архітекторам замінити кілька легко завантажених машин меншою кількістю більш сильно завантаженою машинами, щоб мінімізувати SWAP-C (розмір, вага, потужність та охолодження), звільнити апаратне забезпечення для нових функціональних можливостей, підтримки балансування навантаження та підтримки хмарних обчислень, ферм серверів та мобільних обчислень;

– підвищення надійності – модульність та ізоляція, що забезпечуються віртуалізацією на рівні операційної системи, покращує надійність та експлуатаційну готовність, локалізуючи вплив дефектів між контейнерами, які запущено паралельно;
– масштабованість – один контейнерний механізм може ефективно управляти великою кількістю контейнерів, дозволяючи створювати додаткові контейнери за необхідності;
– ізольованість – контейнери підтримують полегшену просторову ізоляцію, надаючи кожному контейнеру власні ресурси (наприклад, кількість процесорного часу, пам'ять та доступ до мережі) та специфічні для контейнера простори імен;
– продуктивність – у порівнянні віртуальними машинами, контейнери підвищують продуктивність (пропускну здатність), оскільки вони не імітують базове обладнання;
– портативність – оскільки контейнерні зображення набагато менші, ніж віртуальні машини, їх легше передавати та економити простір у файловій системі хоста. Віртуальні машини, навпаки, повинні мати копію всієї операційної системи, включаючи ядро, системні бібліотеки, файли конфігурації, усі каталоги, необхідні операційній системі та всі утиліти. Це різко збільшує розмір зображення і не так просто ділитися ним. Зображеннями контейнерів можна обмінюватися будь-якими способами, а в інтернеті існує ряд центрів обміну зображеннями контейнерів. Зображення

віртуальних машин не мають таких централізованих концентраторів і, як правило, їх потрібно завантажувати на інший сервер;

– пришвидшений цикл розробки – контейнери дуже підходять для короткострокових потреб у застосуванні, оскільки їх можна швидко встановити, переносити та запускати набагато швидше. Однак вони обмежені відсутністю спеціальної операційної системи, ресурсів обробки та зберігання. Контейнери слід використовувати, коли найбільшим пріоритетом є максимізація кількості програм, що працюють на мінімальній кількості серверів. Однак віртуальні машини набагато краще підходять для програм, які потрібно використовувати протягом тривалого періоду часу, оскільки вони працюють у віртуалізованому середовищі, яке є більш надійним та універсальним.

Незважаючи на те, що є багато переваг для переходу на контейнерну віртуалізацію, вони також мають і наступні недоліки:

- спільні ресурси – додатки в контейнерах мають багато ресурсів, включаючи ресурси, специфічні для контейнера, включаючи контейнер, механізм контейнерів та ядро ОС. Через те, що так багато спільних ресурсів, то вони є єдиною точкою відмови;
- аналіз перешкод – спільні ресурси передбачають втручання. Віртуалізація за допомогою контейнерів збільшує складність аналізу тимчасових перешкод і, як правило, робить отримані оцінки часу надто консервативними. Аналіз перешкод стає більш складним, оскільки кількість контейнерів збільшується, а віртуалізація через контейнери поєднується з віртуалізацією через віртуальні машини та багатоядерною обробкою. Кількість інтерференційних шляхів швидко зростає із збільшенням кількості контейнерів. Отримана велика кількість інтерференційних шляхів зазвичай унеможливлює вичерпний аналіз усіх таких шляхів.

Зазвичай, життєвий цикл програмного продукту із використанням технологій контейнерної віртуалізації включає в себе два етапи:

- етап створення зображення контейнера

- безпосередній запуск контейнера з його образу.

Етапи життєвого циклу продукту із використанням технологій базової віртуалізації на базі найпопулярнішої системи контейнерної віртуалізації Docker [5] зображено на рисунку 1.6. Із файлу опису контейнера створюється його зображення, а вже з зображення створюється безпосередньо контейнер.

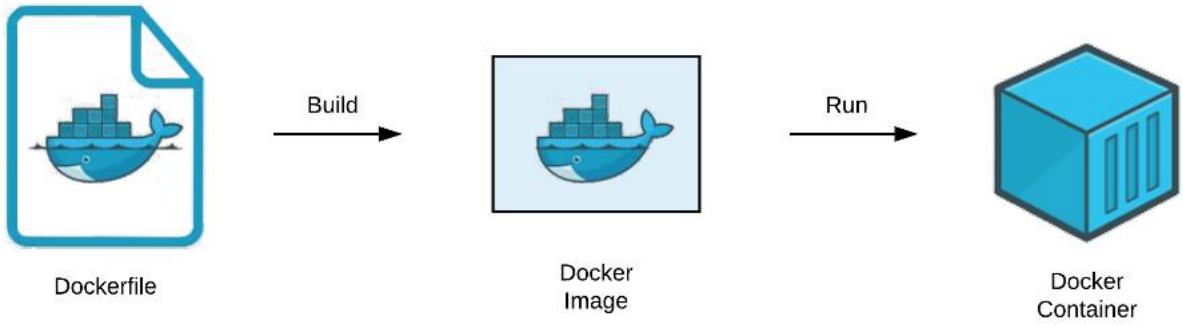


Рисунок 1.6 – Етапи життєвого циклу продукту із використанням технологій контейнерної віртуалізації

Перший етап включає в себе опис змісту контейнера, та команд що там мають виконуватися, тобто, створення зображення контейнера. Зображення контейнера – це незмінний файл, що містить вихідний код, бібліотеки, залежності, інструменти та інші файли, необхідні для запуску програми.

Через те, що ці зображення найчастіше мають режим тільки для читання, їх іноді називають снапшотами. Вони представляють програму та її віртуальне середовище в певний момент часу. Це дозволяє розробникам тестувати та експериментувати з програмним забезпечення у стабільних, однакових умовах.

Оскільки зображення, певним чином, є лише шаблонами, їх не можна запускати. Для цього, необхідно зібрати та запустити контейнер. Зрештою, контейнер - це просто запущене зображення. Після створення контейнера він додає шар для запису поверх незмінного зображення, тобто тепер його можна змінити. Схему запуску контейнера на хості зображено на рисунку 1.7.

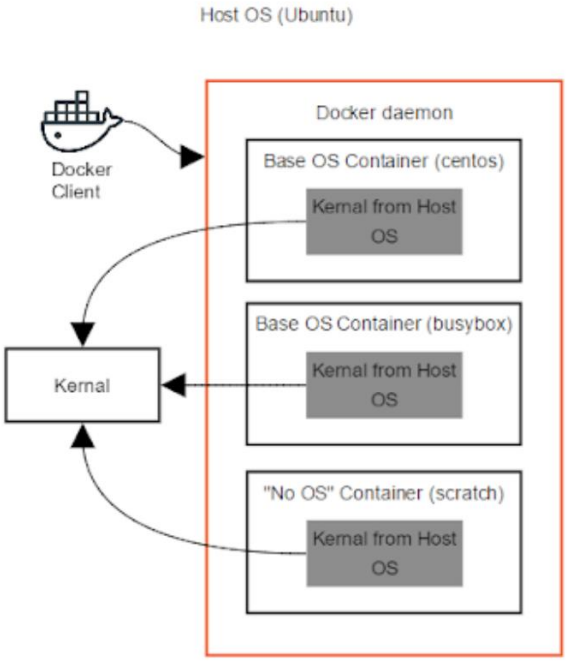


Рисунок 1.7 – Схема запуску контейнера на хості

Образ, на основі якого створюється контейнер, існує окремо і не може бути змінений. Коли запускається контейнерне середовище, по суті, створюється копія цієї файлової системи для читання-запису всередині контейнера. Це додає шар контейнера, який дозволяє модифікувати всю копію зображення.

1.3 Відмінності віртуальних машин від контейнерів

Як вже було наведено в минулих підрозділах, існує два основних підходи для створення декількох ізольованих просторів для виконання програм на одному фізичному сервері: створення віртуальних машин, або використання контейнерної віртуалізації.

Незважаючи на те, що обидва методи є методами віртуалізації, вони мають багато концептуальних відмінностей, через які використання того чи іншого методу є сумлінним вибором кінцевого користувача.

Відмінності між віртуальними машинами та контейнерами наведено у таблиці 1.1.

Таблиця 1.1 – Відмінності між віртуальними машинами та контейнерами.

Властивість	Віртуальні машини	Контейнери
Ізоляція	Забезпечує повну ізоляцію від головної операційної системи та інших віртуальних машин. Це корисно, коли сильний кордон безпеки є критично важливим, наприклад, розміщення програм від конкуруючих компаній на одному сервері або кластері	Зазвичай забезпечує легку ізоляцію від хоста та інших контейнерів, але не забезпечує настільки сильних меж безпеки, як віртуальна машина. Зазвичай рівень безпеки контейнеру можна підвищити шляхом поміщення контейнера у легку віртуальну машину
Операційна система	Запускає повну операційну систему, включаючи ядро, вимагаючи, таким чином, більше системних ресурсів (ЦП, пам’ять та пам’ять)	Запускає частину режиму користувача в операційній системі і може бути адаптована до вмісту лише необхідних служб для вашої програми, використовуючи менше системних ресурсів
Мережа	Використовує віртуальні мережеві адаптери	Використовує ізольований вигляд віртуального мережевого адаптера
Версіонування	Відсутні ефективні механізми для знаходження різниці між двома віртуальними машинами	Зображеня контейнерів можна легко аналізувати на різницю: нові та видалені строки у декларації зображення контейнера

Продовження таблиці 1.1

Сумісність для гостей	Запускає практично будь-яку операційну систему всередині віртуальної машини	Працює на тій самій версії операційної системи, що і хост (ізоляція дозволяє запускати попередні версії тієї самої ОС в полегшеному середовищі віртуальної машини)
Розгортання	Розгортати окремі віртуальні машини за допомогою спеціалізованих програмних застосунків. Швидкість розгортання найчастіше займає декілька хвилин.	Можливо розгортати окремі контейнери за допомогою Docker через командний рядок; розгортати кілька контейнерів за допомогою оркестратора. Контейнер розгортається за лічені секунди.
Відмовостійкість	Віртуальні машини можна пернести на інший сервер кластера, при цьому операційна система віртуальної машини перезапуститься на новому сервері	Контейнер можна швидко перезапустити із його зображення, як на поточному сервері, так і на іншому

Незважаючи на те, що існує ще багато причин використовувати віртуальні машини, контейнери забезпечують рівень гнучкості та портативності, який ідеально підходить для мультиоблачного світу. Коли розробники створюють нові програми, вони можуть не знати всіх місць, де його потрібно буде розгорнути. Сьогодні організація може запустити додаток у своїй приватній хмарі, але завтра може знадобитися розгорнути його у загальнодоступній хмарі від іншого

постачальника. Контейнеризація програм забезпечує командам гнучкість, необхідну для роботи з багатьма програмними середовищами сучасних IT.

Отже, основною різницею між контейнерами та віртуальною машиною є віртуалізація операційної системи та обладнання відповідно. Під час використання віртуальної машини кожна віртуальна машина має власну базову операційну систему, тоді як, використовуючи контейнери, кожен контейнер працює на одному базовому екземплярі операційної системи.

1.4 Обмеження ресурсів контейнера

Однією з важливих особливостей використання контейнерної віртуалізації є можливість обмежити ресурси окремого контейнера. Така особливість є важливою через низку причин:

- створення обмежень для контейнерів, щоб один один або група контейнерів не зайняли усі обчислювальні ресурси;
- можливість планування ресурсної ємності фізичних або віртуальних машин для запуску декількох контейнерів;
- можливість планування ресурсної ємності кластера ресурсів для заданого контейнерного навантаження.

У даному підрозділі описані основні методи регулювання ресурсів контейнера на базі системи управління контейнерами Docker. Методи регуляції для інших систем є достатньо схожими: використовуються подібні інструкції та принципи роботи із ресурсами.

Під ресурсами розуміються чотири складові: ресурси CPU, кількість RAM, об’єм дискового простору та мережеві ресурси.

За замовчуванням контейнер не має обмежень ресурсів і може використовувати стільки даного ресурсу, скільки дозволяє планувальник ядра хоста. Docker надає способи контролювати, скільки пам'яті або процесора може використовувати контейнер, встановлюючи прапорці конфігурації виконання

команди docker run. Цей підрозділ містить детальну інформацію про те, коли слід встановлювати такі обмеження та можливі наслідки їх встановлення.

Важливо не допустити, щоб запущений контейнер споживав занадто багато пам'яті хост-машини. На ядрах Linux, якщо ядро виявляє, що недостатньо пам’яті для виконання важливих системних функцій, воно видає Out Of Memory (OOM) Exception і починає припиняти процеси, щоб звільнити пам’ять. Будь-який процес підлягає припиненню, включаючи Docker та інші важливі програми. Це може ефективно зруйнувати всю систему, якщо буде припинено неправильний процес.

Docker намагається пом'якшити ці ризики, регулюючи пріоритет OOM на демоні Docker, щоб він мав менше шансів бути вбитим, ніж інші процеси в системі. Пріоритет OOM на контейнерах не коригується. Це робить більш імовірним припинення окремого контейнера, ніж демона Docker чи інших системних процесів.

Можна зменшити ризик нестабільності системи через OOME, виконавши наступні рекомендації:

- виконати тести, щоб зрозуміти вимоги до пам'яті додатка, перш ніж виводити його у реальний світ;
- переконатися, що контейнер працює лише на хостах з достатніми ресурсами;
- обмежте обсяг пам'яті, яку може використовувати контейнер;
- приділяти особливу увагу, при налаштуванні свапу. Свап є повільнішим і менш продуктивним, ніж пам'ять, але може забезпечити буфер проти закінчення системної пам'яті.

Docker може застосовувати жорсткі обмеження пам’яті, які дозволяють контейнеру використовувати не більше заданого обсягу пам’яті користувача або системи, або м’які обмеження, які дозволяють контейнеру використовувати стільки пам'яті, скільки йому потрібно, якщо не виконуються певні умови, наприклад, коли ядро виявляє недостатню пам'ять або суперечки на хост-машині. Деякі з цих параметрів мають різний ефект, коли використовуються окремо або коли встановлено кілька варіантів. У таблиці 1.2 наведено базові команди для обмеження ресурсів контейнера.

Таблиця 1.2 – Команди для обмеження ресурсу пам’яті контейнера

--memory	Максимальна кількість пам’яті виділена для контейнера
--memory-swap	Кількість пам’яті, яку контейнер може вигрузити на диск
--memory-swappines	За замовчуванням ядро хосту може замінювати відсоток анонімних сторінок, що використовуються контейнером. Цей параметр регулює відсоток
--kernel-memory	Максимальний обсяг пам'яті ядра, який може використовувати контейнер

За замовчуванням доступ кожного контейнера до часу процесора головного комп'ютера необмежений. Можливо встановити різні обмеження, щоб обмежити доступ контейнера до процесора хост-машини. Найчастіше використовується CFS у реальному часі.

CFS – це планувальник процесора ядра Linux для звичайних процесів Linux. Кілька прапорів виконання дозволяють налаштувати обсяг доступу до ресурсів процесора, який має контейнер.

У таблиці 1.3 наведено базові команди для обмеження ресурсів процесора для контейнера.

Таблиця 1.3 – Команди для обмеження ресурсів процесора для контейнера

--cpus=<value>	вказує, скільки доступних ресурсів ЦП може використовувати контейнер. Вимірюється в мілісекундах. Тут 1000 позначає одиницю ядра процесора за секунду
--cpu-period=<value>	вказує період планування процессору. Використовується із --cpu-quota. Стандартне значення: 100 мілісекунд. Для більшості випадків використання --cpus є більш зручною альтернативою.

Продовження таблиці 1.3

--cpu-quota	накладає на контейнер квоту для процесора. Позначає кількість мікросекунд за --cpu-period, коли контейнер має доступ до процесора. У інший час контейнер не має доступу до процесора.
--cpuset-cpus	обмеження на окремі CPU чи ядра, які контейнер може використовувати.

Також, можливо обмежувати доступ контейнера до ресурсыв диску. Обмеження можливе за допомогою команд, наведених у таблиці 1.4.

Таблиця 1.4 – Команди для обмеження ресурсів диску для контейнера

--device-read-bps	ліміт швидкості читання з диску
--device-write-bps	ліміт швидкості запису на диск
--device-read-iops	ліміт кількості операцій читання з диску за секунду
--device-write-iops	ліміт кількості операцій запису на диск за секунду

Отже, аналізуючи дані конфігурації можна дійти висновку, що контейнери можна обмежити в усіх основних напрямках з приводу ресурсів, що дозволяє більш точно керувати можливою кількістю контейнерів на одному хості та більш точно прогнозувати необхідну надалі кількість хостів, тобто, планувати ресурсну ємність для заданого навантаження.

1.5 Висновки до розділу

Поява віртуалізації дозволила більш ефективно використовувати ресурси фізичної машини: процесорний час, кількість оперативної пам’яті, дисковий простір, мережеві ресурси більш ефективно за рахунок створення всередині однієї фізичної машини декількох віртуальних системи, які виконуються у різних, незалежних середовищах, які не мають вплив одне на одного. Основним методом

використання віртуалізації було створення віртуальних машин. Проте, сьогодні, на заміну класичним віртуальним машинам приходять технологія віртуалізації на рівні операційної системи, тобто, контейнерна віртуалізація.

Контейтерна віртуалізація має достатньо широкий ряд переваг через відмінний від класичного тип віртуалізації: вітуалізується не апаратне забезпечення, що вимагає повноцінних віртуалізованих образів операційної системи, а лише операційна система. Через це, зменшуються накладні витрати на підтримку віртуалізації.

Контейнери мають широкий набір інструкції для керування кількості використовуваних ресурсів хоста, що є необхідною умовою для планування ресурсної ємності.

2 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ДЛЯ ОРКЕСТРАЦІЇ КОНТЕЙНЕРІВ

Перед розробниками найчастіше постає проблема керування багатьма контейнерами одночасно, що є достатньо складною задачею, порівняно із запуском одного, або декількох контейнерів, як було показано у розділі 1. Для задоволення цих потреб виникла технологія як оркестрація контейнерів.

Оркестрація контейнерів це автоматизована координація, конфігурація, та управління їх сукупністю. Оркестрацію контейнерів виконують так звані контейнерні окрестратори. Їх основною задачею є опис того, як контейнери взаємодіють у системі, підтримка їх життєвого циклу, провізія ресурсів, забезпечення ресурсної ємності, автоматичне оновлення контейнерів тощо.

Найяскравішими представниками контейнерних оркестраторів є Docker Swarm, Nomand, AWS ECS та Kubernetes. Хоча такі системи і мають відмінні структурні компоненти та алгоритми, вони виконують схожу задачу – оркеструють контейнерне навантаження.

Оскільки більша частина бізнесу використовує контейнерні оркестратори, тому найкращим науково-практичним результатом планування ресурсної ємності для заданого навантаження буде розроблення такого алгоритму саме для однієї з систем оркестрації контейнерами.

У даному розділі розглянуто основні особливості таких систем як Docker Swarm, Nomand, AWS ECS та Kubernetes, порівняно та проаналізовано дані системи. На базі аналізу обрано цільову систему для планування ресурсної ємності.

2.1 Docker Swarm

Docker Swarm - це група фізичних або віртуальних машин, на яких запущена програма Docker і які були налаштовані на об’єднання в кластер [5]. Після того, як група машин буде згрупована, все ще можна запускати команди Docker, але тепер вони будуть виконуватися машинами у всьмоу кластері. Діяльність кластера

контролюється менеджером swarm, а машини, які приєдналися до кластера, називаються вузлами. Архітектуру Docker Swarm показано на рисунку 2.1.

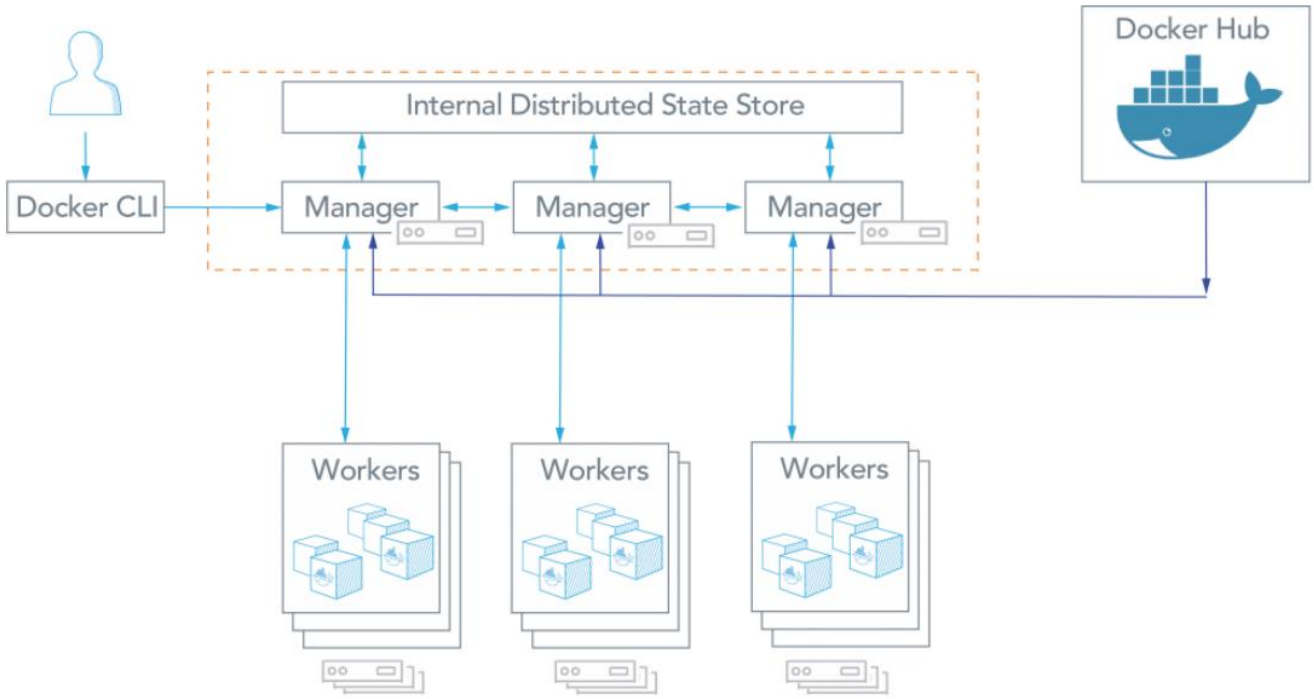


Рисунок 2.1 – Архітектура Docker Swarm

Docker swarm – це інструмент для організації контейнерів, що означає, що він дозволяє користувачеві керувати кількома контейнерами, розгорнутими на декількох хост-машинах.

Однією з ключових переваг, пов’язаних з роботою docker swarm, є високий рівень доступності програм, тобто, контейрерів. У docker swarm зазвичай є кілька робочих вузлів і принаймні один менеджерський вузол, який відповідає за ефективну обробку ресурсів робочих вузлів та забезпечення ефективної роботи кластера.

Docker swarm складається з групи фізичних або віртуальних машин, що працюють у кластері. Коли машина приєднується до кластера, вона стає вузлом у Docker swarm. Docker swarm має три різні типи вузлів, кожен із різною роллю в екосистемі:

- Manager node – основною функцією менеджерських вузлів є призначення завдань робочим вузлам у swarm. Вузли менеджера також допомагають виконувати деякі управлінські завдання, необхідні для управління кластером. Docker рекомендує максимум сім вузлів менеджера для Docker swarm;
- Leader node – Коли кластер встановлений, алгоритм консенсусу Raft [6] використовується для призначення одного з них як «лідерного вузла». Ведучий вузол приймає всі рішення щодо управління кластером та бере на себе роль оркестру. Якщо вузол лідера стає недоступним через відключення або збій, новий вузол лідера може бути обраний за допомогою алгоритму консенсусу Raft;
- Worker node – У Docker swarm з численними хостами кожен робочий вузол функціонує, отримуючи та виконуючи завдання, які йому присвоюють вузли менеджера. За замовчуванням усі режими менеджера також є робочими вузлами і можуть виконувати завдання, коли у них є для цього доступні ресурси.

Все більша кількість розробників, використовують Docker і Docker swarm, щоб ефективніше створювати, оновлювати та експлуатувати програми. Навіть такі гіганти програмного забезпечення, як Google, застосовують методології, засновані на контейнерах, такі як Docker Swarm. Наступні причини сприяють поширенню популярності Docker Swarm:

- використання потужностей контейнерів – повністю використовуються конструктивні переваги, які пропонують контейнери. Контейнери дозволяють розробникам розгортати додатки або служби в автономних віртуальних середовищах - завдання, яке раніше було доменом віртуальних машин. Контейнери демонструють більш полегшену версію віртуальних машин, оскільки їх архітектура дозволяє ефективніше використовувати обчислювальну потужність;
- висока доступність – збільшення доступності додатків через надмірність. Для того щоб функціонувати, docker swarm повинен мати manager node, який може призначати завдання робочим вузлам. Застосовуючи декілька

менеджерів, розробники гарантують, що система може продовжувати функціонувати, навіть якщо один із вузлів менеджера вийде з ладу;

– автоматичне балансування навантаження – Docker swarm планує завдання, використовуючи різні методології, щоб переконатися, що для всіх контейнерів достатньо ресурсів.

2.2 AWS ECS

AWS ECS (Amazon Web Services Elastic Container Service) – це масштабована, швидка служба управління контейнерами, яка дозволяє легко запускати, зупиняти та керувати контейнерами на кластері [7]. Контейнери визначені у декларації завдання, яке можливо використовувати для запуску окремих завдань або завдань у службі. У цьому контексті служба – це конфігурація, яка дозволяє одночасно запускати та підтримувати певну кількість завдань у кластері. Можливо запускати завдання та служби на безсерверній інфраструктурі, якою керує AWS Fargate [8]. Крім того, для більшого контролю над власною інфраструктурою можливо запускати завдання та служби на кластері екземплярів Amazon EC2.

Amazon ECS дозволяє запускати та зупиняти додатки на основі контейнерів за допомогою простих викликів API. Також можливо отримати стан кластера з централізованої служби та мати доступ до багатьох знайомих функцій Amazon EC2.

Структуру Amazon ECS із використанням Amazon EC2 для створення кластера наведено на рисунку 2.2.

У Amazon ECS є можливість запланувати розміщення контейнерів у власному кластері на основі власних потреб у ресурсах, політики ізоляції та вимог щодо доступності. За допомогою Amazon ECS не потрібно керувати власними системами управління кластерами та управління конфігурацією або турбуватися про масштабування інфраструктури управління.

Amazon ECS може бути використаний для створення послідовного процесу збірки та розгортання, управління та масштабування робочих навантажень

пакетного та потокового навантаження, та побудови складної архітектури додатків на моделі мікросервісів.

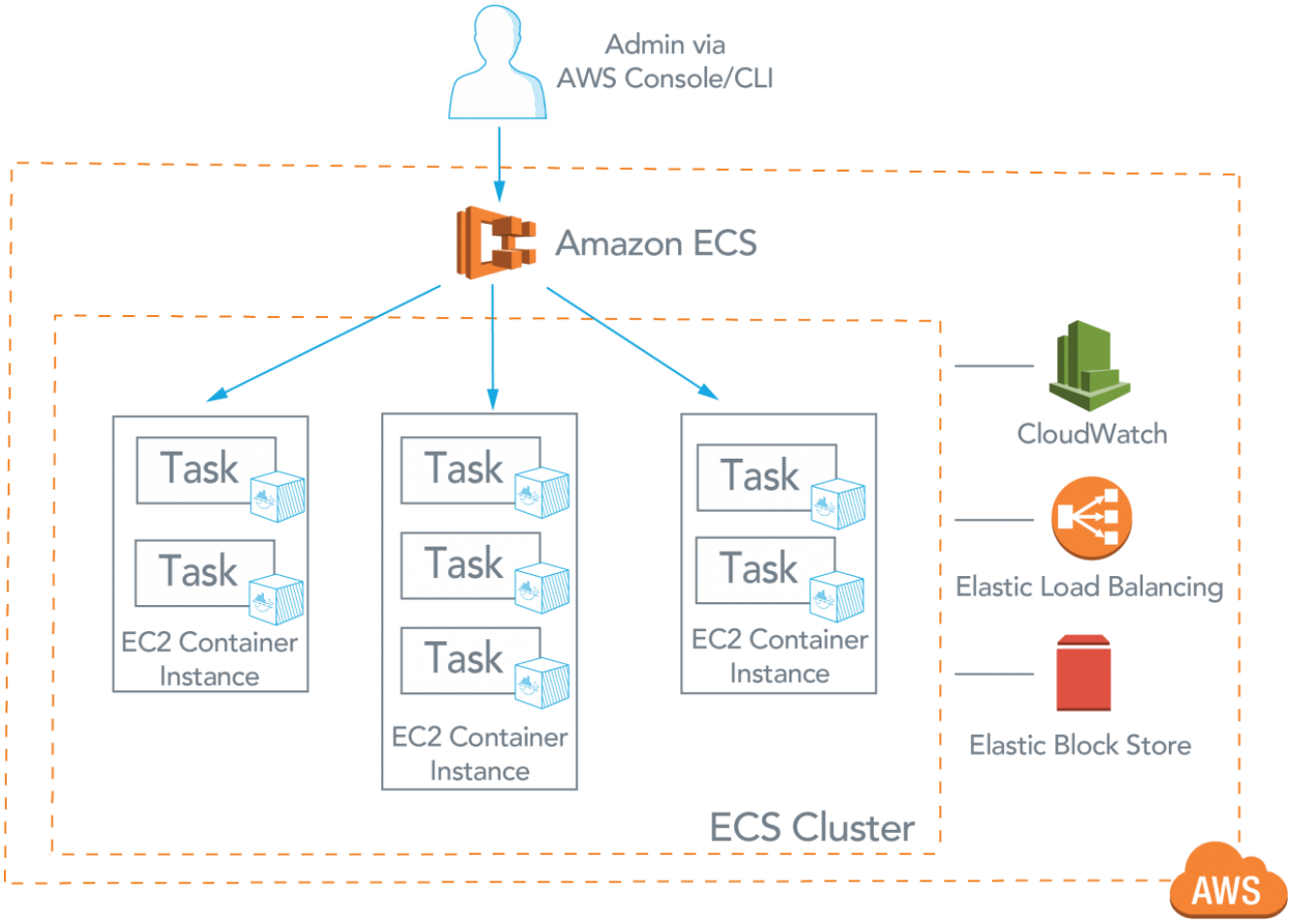


Рисунок 2.2 – Архітектура Amazon ECS із використанням EC2 для створення кластера

Amazon ECS – це регіональний сервіс, який спрощує запуск контейнерів у високодоступному режимі в декількох зонах доступності в регіоні. Можливо створити кластери Amazon ECS в рамках нового або існуючого VPC. Після запуску кластера можна створити визначення завдань, які визначають, які образи контейнерів запускатимуться у кластері. Визначення ваших завдань використовуються для запуску завдань або створення служб. Зображення контейнерів зберігаються та витягуються з реєстрів контейнерів, наприклад, Amazon Elastic Container Registry.

Щоб підготувати програму до запуску на Amazon ECS, потрібно створити визначення завдання. Визначення завдання – це текстовий файл (у форматі JSON), що описує один або декілька контейнерів (максимум до десяти), які формують програму. Визначення завдання можна сприймати як проект програми. Він визначає різні параметри для програми. Наприклад, ці параметри можна використовувати, щоб вказати, які контейнери слід використовувати, які порти слід відкрити для програми та які обсяги даних слід використовувати з контейнерами в завданні. Конкретні параметри, доступні для визначення завдання, залежать від потреб конкретного додатка.

Завдання у Amazon ECS – це результат створення екземпляра визначення завдання всередині кластера. Після того, як створено визначення завдання для додатка в Amazon ECS, можна вказати кількість завдань, які запускатимуться у кластері.

Кластер Amazon ECS – це логічне групування завдань або послуг Можливе зареєструвати один або кілька екземплярів Amazon EC2 (їх також називають екземплярами контейнерів) у своєму кластері, щоб запускати на них завдання. Або можливо використовувати безсерверну інфраструктуру, яку надає Fargate, для запуску завдань. Коли завдання виконуються на Fargate, ресурсами кластера також керує Fargate.

Коли вперше використовується Amazon ECS, створюється кластер за замовчуванням. Є можливість створити додаткові кластери в обліковому записі, щоб ресурси були окремими.

Агент контейнерів працює на кожному вузлі в кластері Amazon ECS. Агент надсилає інформацію про поточні запущені завдання та використання ресурсів Amazon ECS. Він запускає та зупиняє завдання кожного разу, коли отримує запит від Amazon ECS.

Отже, Amazon ECS є достатньо багатофункціональним механізмом для оркестрації контейнерного навантаження на базі кластера ресурсів, який можна сформувати власноруч.

2.3 Nomad

Nomad [9] – це простий та гнучкий організатор робочих навантажень для розгортання та керування контейнерами (наприклад, Docker), неконтейнерними програмами (виконуваний файл, Java) та віртуальними машинами (qemu) у локальній та хмарній областях.

Nomad підтримується в Linux, Windows та macOS. Також доступна комерційна версія Nomad, Nomad Enterprise. Архітектурні компоненти Nomand показано на рисунку 2.3.

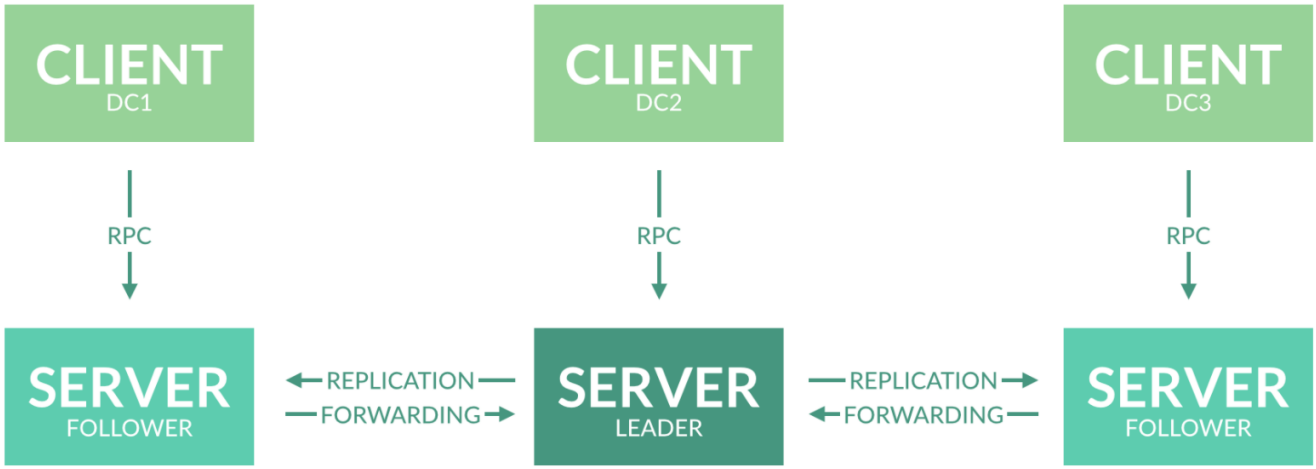


Рисунок 2.3 – Архітектурні компоненти Nomand

Основними властивостями Nomad є:

– можливість розгортати конейнери – гнучкість Nomad як оркестратора дозволяє організації запускати контейнери, застарілі та пакетні програми разом на одній і тій же інфраструктурі. Nomad приносить основні переваги оркестрації застарілим програмам, не потребуючи обмеження за допомогою підключаються драйверів завдань;
– простота використання – Nomad працює як єдиний двійковий файл і повністю автономний – поєднує управління кластером та запуск задач в єдину систему. Nomad не вимагає жодних зовнішніх служб для зберігання чи координації. Nomad автоматично обробляє помилки програм, вузлів та

драйверів. Nomad розподілений та стійкий, використовуючи вибори лідера та реплікацію штатів, щоб забезпечити високу готовність у разі невдач;

– пропускна здатність – Nomand має високу пропускну здатність та низьку затримку. Може підтриувати до 10 тисяч робочих вузлів;
– можливість підтримки декілької регіонів – Nomad був розроблений для підтримки інфраструктури в глобальному масштабі. Nomad підтримує багаторегіональність і може розгортати програми в різних регіонах.

У Nomand надано велику кількість структурних компонентів, кожен з яких відіграє важливі роль у загальній інфраструктурі. Перелік основних структурних компонентів у Nomand:

– одиниця роботи – це специфікація, надана користувачами, яка декларує корисне навантаження для Nomad. Одиниця роботи – це форма бажаного стану; користувач декларує, що завдання має виконуватися, але не там, де його слід виконувати; Обов'язок Nomad полягає в тому, щоб переконатися, що фактичний стан відповідає бажаному станом користувача. Одиниця роботи складається з однієї або декількох груп завдань;
– завдання – це найменша одиниця роботи в Nomad. Завдання виконуються драйверами, що дозволяє Nomad бути гнучким у типах завдань, які він підтримує. Завдання визначають їх драйвер, конфігурацію драйвера, обмеження та необхідні ресурси;
– розподілення – це відображення між групою завдань у одиниці роботі та вузлом клієнта. Одна одиниця роботи може мати сотні або тисячі груп завдань, тобто еквівалентна кількість розподілів повинна існувати для відображення роботи на клієнтських машинах. Розподіли створюються серверами Nomad як частина планування рішень, прийнятих під час оцінки;
– клієнт – у Nomad це машина, на якій можна запускати завдання. Усі клієнти використовують агент Nomad. Агент відповідає за реєстрацію на серверах, спостереження за будь-якою роботою, яка буде призначена, та виконання завдань. Агент Nomad – це тривалий процес, який взаємодіє із серверами;

– сервер – мозок кластера. Існує кластер серверів для кожного регіону, і вони управляють усіма завданнями та клієнтами, виконують оцінки та створюють розподіл завдань. Сервери копіюють дані між собою та проводять вибори лідера, щоб забезпечити високу доступність. Сервери об'єднуються в різні регіони, щоб зробити Nomad глобально обізнаним.

У кожному регіоні є як клієнти, так і сервери. Сервери відповідають за прийняття завдань від користувачів, управління клієнтами та обчислення розміщення завдань. У кожному регіоні можуть бути клієнти з декількох центрів обробки даних, що дозволяє невеликій кількості серверів обробляти дуже великі кластери.

2.4 Kubernetes

Kubernetes [10] – це портативна, розширювана платформа з відкритим кодом для управління робочими навантаженнями та послугами в контейнерах, що полегшує як декларативну конфігурацію, так і автоматизацію. Він має велику, швидко зростаючу екосистему. Послуги, підтримка та інструменти Kubernetes широко доступні. Згідно із дослідженням [11], Kubernetes є найпопулярнішим конейнерним оркестратором.

Основними перевагами Kubernetes є:

– підтримка сервісів та автоматичне белансування навантаження – Kubernetes може виставити контейнер, використовуючи ім'я DNS або використовуючи власну IP-адресу. Якщо трафік до контейнера високий, Kubernetes може завантажувати баланс і розподіляти мережевий трафік таким чином, щоб розгортання було стабільним;
– оркестрація сховищ – Kubernetes дозволяє автоматично монтувати обрану систему зберігання, наприклад, локальні сховища, загальнодоступні хмарні провайдери тощо;
– автоматичне розгортання та відкат версій – можливо описати бажаний стан для розгорнутих контейнерів за допомогою Kubernetes, і він може змінити

фактичний стан до бажаного стану з контрольованою швидкістю. Наприклад, можна автоматизувати Kubernetes для створення нових контейнерів для деплойменту, видалення існуючих контейнерів тощо;

– автовідновлення та надійність – Kubernetes перезапускає контейнери, які виходять з ладу, замінює контейнери, вбиває контейнери, які не відповідають на визначену користувачем перевірку стану здоров'я, і не розгортає їх клієнтам, поки вони не будуть готові за заданими критеріями;
– підтримка збереження конфігурацій та збереження чутливої інформації – Kubernetes дозволяє зберігати та керувати конфіденційною інформацією, такою як паролі, токени OAuth та ключі SSH. Можливо розгортати та оновлювати секрети та конфігурацію програми, не відновлюючи образи контейнера та не відкриваючи секрети у конфігурації стека.

Кластер Kubernetes складається з набору робочих машин, званих вузлами, які запускають контейнерні програми. Кожен кластер має принаймні один робочий вузол. Діаграму Kubernetes кластера із його компонентами показано на рисунку 2.4.

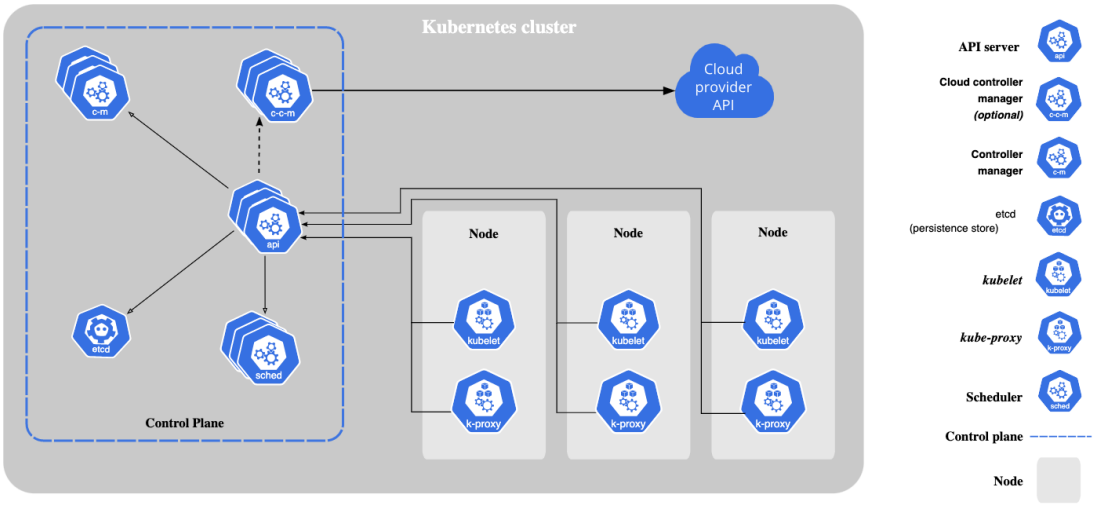


Рисунок 2.4 – Діаграма Kubernetes кластера із його компонентами

Однією з важливих особливостей Kubernetes є наявність користувацького інтерфейсу, що значно полегшує користування даною системою та значно підвищує інформативність. На користувацькому інтерфейсі наведено основні

елементи кластера та усі поточні корисні навантаження. Приклад користувацького інтерфейсу наведено на рисунку 2.5.

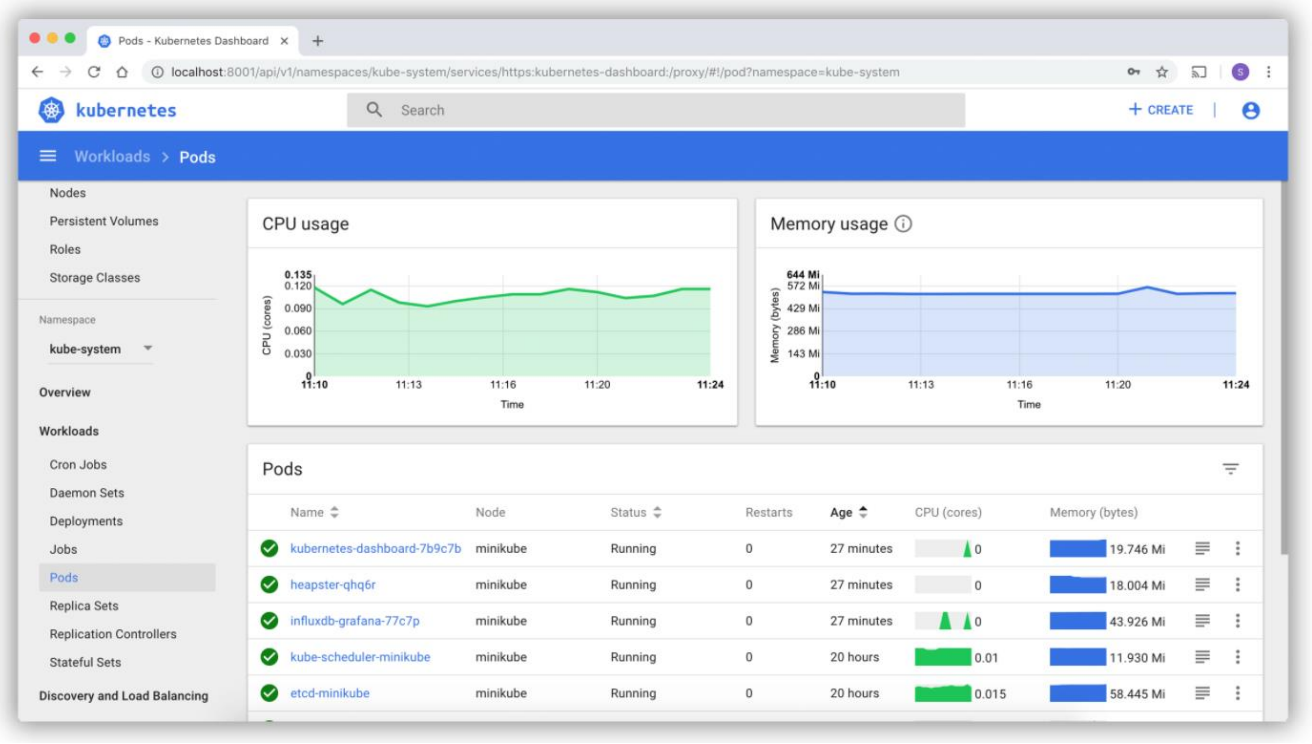


Рисунок 2.5 – Приклад користувацького інтерфейсу Kubernetes

Робочі вузли розміщують поди, які є компонентами робочого навантаження програми. Головний вузол керування управляє робочими вузлами та подами в кластері. У виробничих середовищах головний вузол, як правило, працює на декількох комп'ютерах, а кластер, як правило, працює на декількох вузлах, забезпечуючи відмовостійкість і високу доступність.

2.5 Вибір цільового оркестратора

У попередніх розділах проаналізовано такі контейнерні окрестратори як Docker Swarm, Amazon ECS, Nomad та Kubernetes. У ролі цільової платформи для планування ресурсної ємності слід вибирати найкращий конейнерний окрестратор серед запропонованих, та той, що найчастіше використовується на ринку. Кожен із них має як свої переваги, так і недоліки, проте всі вони мають схожу архітектуру: базуються на кластерах.

Наразі, Kubernetes є найпопулярнішим контейнерним оркестратором, має широку підтримку у суспільства, а також, що не менш важливо, має відкритий вихідний код.

Аналізуючи оркестратор Nomand відносно Kubernetes, можна дійти висновку що у Nomand:

- відсутній аналіз сервісів, тому потрібен додатковий інструмент, наприклад Consul. Consul повинен бути встановлений поряд із Nomad на кожному хості кластера;
- Nomad не має управління чутливою інформацією або іншого спеціального сховища ключів / значень. Інтеграція з Vault або Etcd необхідна для збереження секретів додатків та будь-яких довільних даних, необхідних вашим службам розподілу;
- у відкритій версії Nomad відсутній графічний інтерфейс або інформаційна панель, як ті, що пропонуються Kubernetes. Це обмежує вашу можливість швидко оглянути стан хостів та послуг. Плюс, усім має керувати автоматизований сценарій або інтерфейс командного рядка.

Docker swarm також має деяку кількість мінусів, порівняно із системою Kuberntes, а саме:

- як і у Nomad, Docker Swarm не має корисного інтерфейсу, за винятком кількох базових візуалізаторів, розроблених спільнотою, які лише показують, який контейнер розгорнуто в якому режимі, чого недостатньо для повноцінного аналізу;
- Docker Swarm потребує зовнішнього сервера для зберігання записів виявлення сервісів. Поточні програми, які підтримує Docker Swarm для вирішення цієї задачі: Consul, Etcd, ZooKeeper та текстові файли.

Amazon ECS також має деяку кількість мінусів, порівняно із Kubernetes:

- завжди потрібно створювати нові визначення завдань замість того, щоб змінювати існуючі, що значно подовшує процес розробки;
- ECS також не має простої опції для розповсюдження оновлень (поступове перерозгортання нового образу Docker);

– складність підтримнки великої кількості мікросервісів – ECS має переваги, якщо використовуються невеликі та прості середовища Docker. Як тільки розгортання масштабується до 20 або більше мікросервісів, зробити все, щоб працювати безперебійно, керування конфігураціями, версіями, відкатами та іншими операціями стає складним.

Kubernetes має широку популярність, може підтримувати дуже велику кількість одночасно розгорнутих под, та, що є дуже важливим, Kubernetes – система з відкритим вихідним кодом та має велику кількість налаштувань та додаткових можливостей. Також, як показано у дослідженні [12], Kubernetes має високу доступність, що дозволяю використовувати його при роботі із системами, які потребують високої доступності.

У таблиці 2.1 наведено основні рекомендації щодо використання того чи іншого оркестратора.

Таблиця 2.1 – Рекомендації щодо використання оркестраторів

Система	Користувацький інтерфейс	Рекомендації для використання
Docker Swarm	Відсутній	Для невеликих кластерів та простої архітектури. Для невеликих команд
Amazon ECS	Присутній	Для невеликої кількості контейнів у кластері. Для навантаженя, які не потребують частого оновлення. Якщо є готові інтеграції із платформою Amazon Web Services
Nomad	Відсутній	Для невеликої кількості контейнерів. Для невеликих, висококваліфікованих команд.

Продовження таблиці 2.1

Система	Користувацький інтерфейс	Рекомендації для використання
Kubernetes	Присутній	Для будь-якого рівня складності контейнерного навантаження та будь-якої їх кількості. Для команд будь-якого розміру із різними ролями та необхідним рівнем доступу

Отже, аналізуючи вищенаведені фактори, цільовою платформою для планування ресурсної ємності є Kubernetes через свою широку популярність та кращу підтримку основних можливостей для оркестрації контейнерів, ніж системи-конкуренти.

2.6 Особливості формування ресурсної ємності та контейнерного навантаження у Kubernetes

Для правильного формування ресурсної ємності для заданого контейнерного навантаження у Kubernetes, необхідно розуміти що формує контейнерне навантаження: тобто, як контейнер представлено у загальній архітектурі та що, безпосередньо, формує ресурсну ємність. Необхідно розуміти можливості керування даними аспектами.

Kubernetes запускає робоче навантаження, розміщуючи контейнери в подах для роботи на вузлах. Вузол може бути віртуальною або фізичною машиною, залежно від кластера. Кожен вузол управляється головним вузлом і містить послуги, необхідні для запуску под [13]. Спрощену архітектурну схему Kubernetes з використанням под та нод наведено на рисунку 2.6. Діаграму компонентів Kubernetes надано у додатку В.

Тобто, кластер Kubernetes переважно складається із вузлів. Серед усіх вузлів є один головний вузол Kubernetes master, або ж головний вузол, який контролює статус усього кластера та надає кінцевому користувачеви API для роботи із Kubernetes.

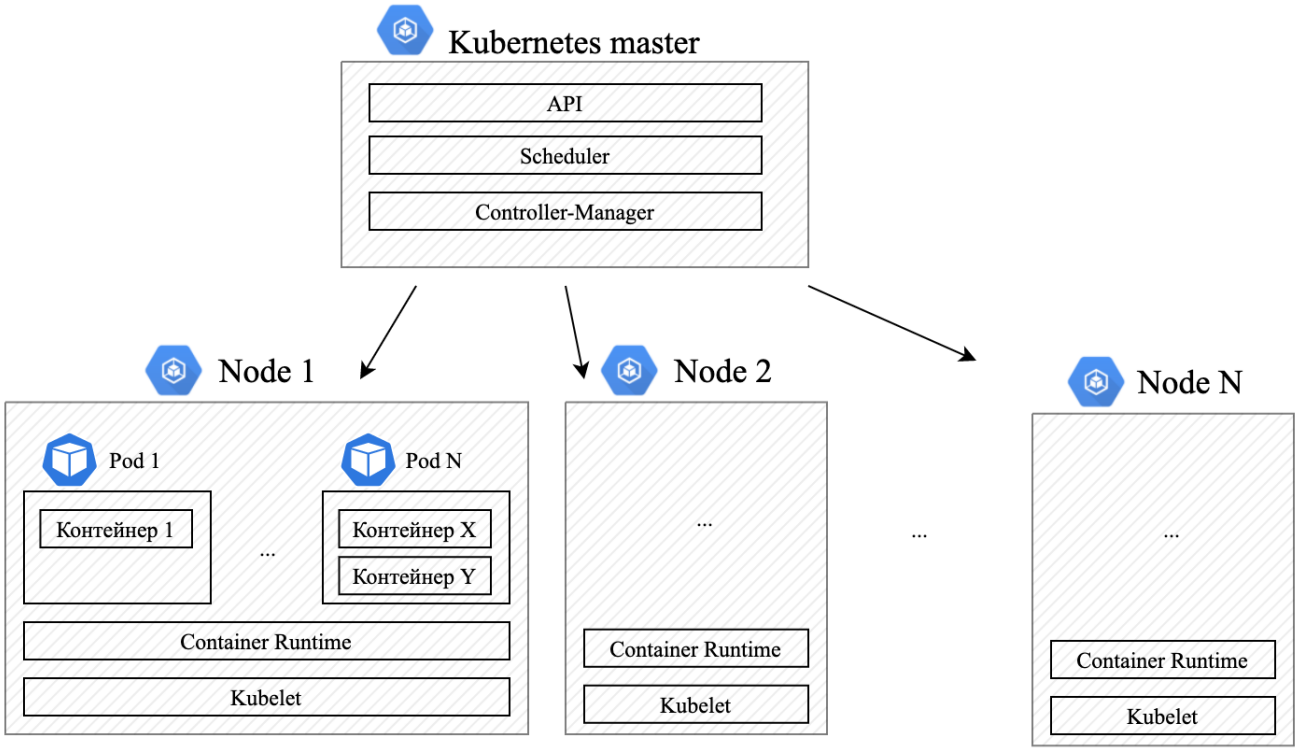


Рисунок 2.6 – Схема Kubernetes із використанням вузлів та под

Поди – це найменші, найосновніші об’єкти для розгортання в Kubernetes. Pod представляє один екземпляр запущеного процесу у кластері.

Поди містять один або кілька контейнерів, таких як контейнери Docker. Коли Pod запускає кілька контейнерів, контейнери управляються як єдине ціле і діляться ресурсами. Як правило, запуск декількох контейнерів в одному поді є розширеним варіантом використання.

Поди у Kubernetes ефемерні. Вони не призначені для запуску довічно, і коли под виключається, його неможливо повернути назад. Найчастіше, поди не зникають, доки їх не видалить користувач або контролер.

Поди не відновлюються самостійно. Наприклад, якщо под запланований на вузлі, який згодом виходить з ладу, под видаляється. Подібним чином, якщо под з будь-якої причини видалено з вузла, под не замінює себе.

Кожен под має об’єкт API PodStatus, який представлений полем стану под. Статус пода – це підсумок високого рівня пода в його поточному стані. Под може знаходитися в одному із нижченаведених статусів:

- очікування – под створено та прийнято кластером, але один або кілька його контейнерів ще не запущені. Ця фаза включає час, витрачений на планування роботи на вузлі та завантаження зображень;
- запуск – под був прив’язаний до вузла, і всі контейнери були створені. Принаймні один контейнер запущений, перебуває в стадії запуску або перезапускається;
- успішно – усі контейнери в Pod закрито успішно;
- помилка: усі контейнери в Pod закінчились, і принаймні один контейнер закінчився зі збоєм;
- невідомо: стан пода визначити неможливо.

Отже, один або декілька контейнерів запускаються у поді, тобто, формують необхідне корисне навантаження. У Kubernetes є можливість керувати ресурсами контейнерів в середині поду. Подібно до того, як біло розглянуто для Docker, можливо керувати основними ресурсами поду: процесорний час та кількість оперативної пам’яті. Можливо вказувати ліміти за ресурсами як для всього пода, так і для кожного контейнера всередині окремо.

Кількість под у Kubernetes необмежена, тому є можливість додавати велику кількість под. Єдиною вимогою є наявність необхідної ресурсної ємності.

Кожен вузол у кластері розміщує поди, які є компонентами робочого навантаження програми. На кожному вузлі мають бути запущені такі процеси як:

- kubelet – агент, який працює на кожному вузлі кластера. Перевіряє, що контейнери працюють у поді. Kubelet бере набір PodSpecs, які надаються за допомогою різних механізмів, і гарантує, що контейнери, описані в цих

PodSpecs, працюють і відповідають заданим критеріям. Kubelet не керує контейнерами, які не були створені Kubernetes.

– container runtime – це програмне забезпечення, яке відповідає за запуск контейнерів. Kubernetes підтримує кілька середовищ виконання контейнерів: Docker, containerd, CRI-O та будь-яку реалізацію Kubernetes CRI (інтерфейс виконання контейнера).

Статус вузла складається з чотирьох основних компонентів:

– адресів – внутрішні та зовнішні IP адреси вузла;
– статус – масив статусів взула. Основою властивістю є властивність Ready, яка вказує на те, чи вузол у робочому стані, та чи готовий вузол приймати нове навантаження, чи ні;
– ємність і доступність;
– загальна інформація – інформація, така як версія ядра, версія Kubernetes, версія Docker тощо.

Важливим показником для ресурсної ємності Kubernetes є властивість «ємність і доступність». Вона описує ресурси, доступні на вузлі: центральний процесор, пам’ять та максимальну кількість под, які можна запланувати на вузол.

Поля в блоці ємності вказують загальну кількість ресурсів, якими володіє вузол. Блок досутпності вказує кількість ресурсів на вузлі, доступних для споживання звичайними подами.

Об'єкти вузла відстежують інформацію про ресурс вузла: обсяг доступної пам'яті та кількість процесорів. Вузли, які самостійно реєструються, повідомляють про свої ресурси під час реєстрації. Якщо вузол додається вручну, то потрібно встановити інформацію про ємність вузла.

Тобто, ресурсна ємність одного вузла визначається кількістю оперативної пам’яті та процесору у даний момент часу. Відповідно, якщо вузли об’єднуються у кластер ресурсів, то ресурсну ємність кластера Kubernetes визначає сукупність ресурсної ємності кожного вузла окремо.

Важливою особливістю Kubernetes є можливість змінювати як кількість под, так і кількість вузлів, що забезпечує, відповідно, горизонтальне масштабування

корисного навантаження та горизонтальне масштабування інфраструктури. Масштабованість у даному розрізі означає можливість додавати до системи нові елементи: поди та вузли. Схему горизонтального масштабування у двох розрізах наведено на рисунку 2.7.

Відповідно до схеми горизонтального масштабування у Kubernetes, є можливість змінювати як корисне навантаження, так і ресурсну ємність усього кластера.

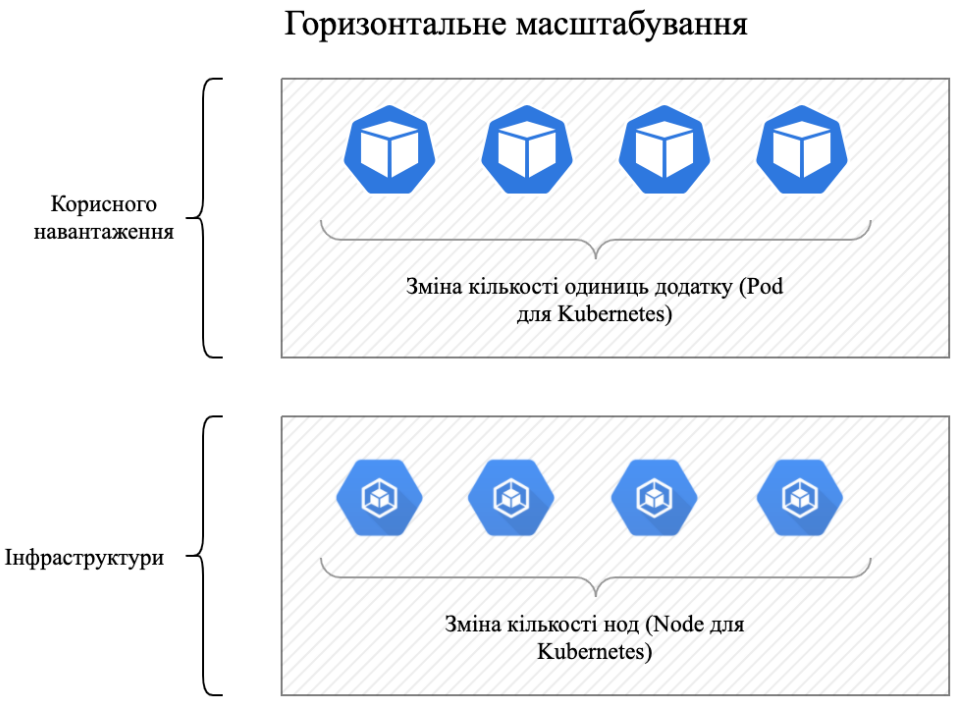


Рисунок 2.7 – Схема горизонтального масштабування у Kubernetes

Також, у Kubernetes є можливість змінювати ресурси об’єктів, а саме: змінити ресурси, необхідні для пода, або ж замінити один вузол таким, що має більшу кількість ресурсів, ніж попередній. Тобто, можливе і вертикальне масштабування. Схему вертикального масштабування наведено на рисунку 2.8.

Отже, у Kuberntes є можливість всесторонньо змінювати ресурси: як ресурси корисного навантаження, так і інфраструктурної сторони – ресурси кластера, тобто, його ресурсну ємність.

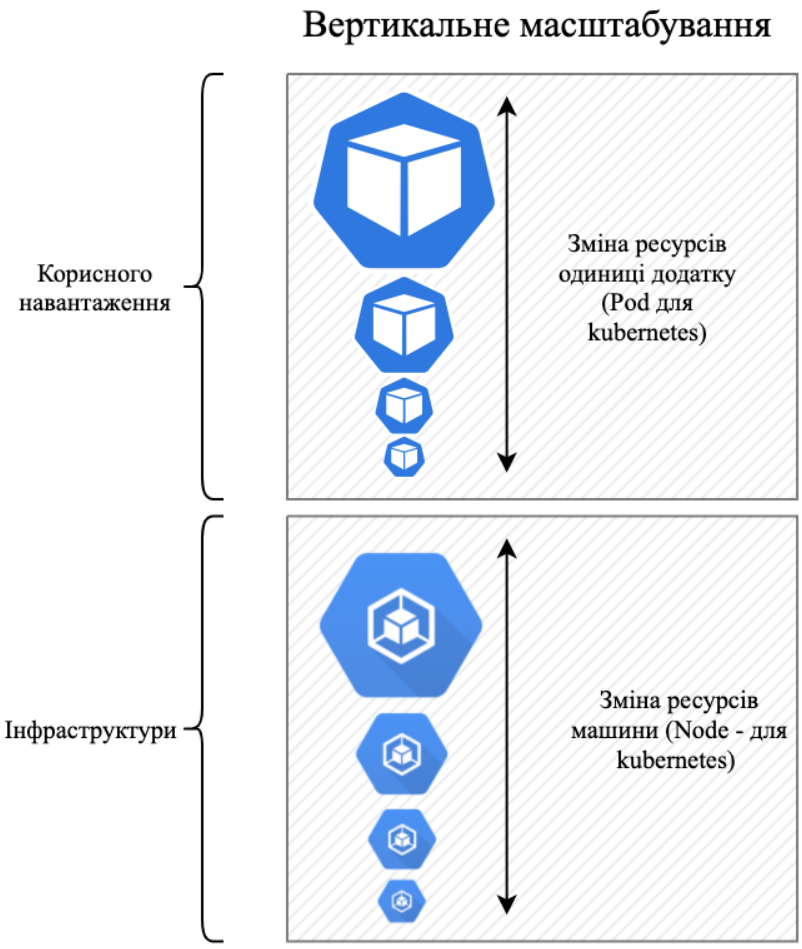


Рисунок 2.8 – Схема вертикального масштабування у Kubernetes

2.7 Висновки до розділу

У даному розділі було обрано цільовий контейнерний оркестратор серед таких оркестраторів: Docker Swarm, Nomand, AWS ECS та Kubernetes. За результатами дослідження та аналізу було обрано систему з відкритим вихідним кодом Kubernetes через її широку популярність, більшу кількість можливостей, порівняно із конурентними системами, високу найдійність, доступність та наявність інтерфейсу користувача.

Для системи Kubernetes описано що формує корисне навантаження та що формує ресурсну ємність.

Корисне навантаження формується за допомогою так званих под – абстракції, яка складається з одного або більше контейнерів. Подібно до того, як було описано

для Docker, можна керувати ресурсами пода. Поди мають бути розташовані на вузлах кластера.

Ресурсна ємність Kubernetes формується із ресурсної ємності його кластера, яка, в свою чергу, визначається як сума ресурсних ємностей усіх вузлів кластера. Під ресурсами вузла мається на увазі його процесор та кількість оперативної пам’яті, вільні для розташування подів.

3 АНАЛІЗ ПРОБЛЕМИ ТА МАТЕМАТИЧНА МОДЕЛЬ ПЛАНУВАННЯ РЕСУРСНОЇ ЄМНОСТІ В УМОВАХ КОНТЕЙНЕРНОЇ ВІРТУАЛІЗАЦІЇ

Алгоритм планування ресурсної ємності та складання математичної моделі є основним підґрунтям для створення вихідного алгоритму. Необхідною умовою є аналіз усіх складових, які впливають на рішення та усіх параметрів, які мають бути вхідними для алгоритму.

У рамках аналізу, цільовою системою для планування ресурсної ємності для заданого навантаження було обрано систему Kubernetes, тому основні аспекти буде наведено саме для цієї системи, проте інформація може бути застосована і для будь-якої іншої системи оркестрації контейнерами, яка базується на моделі роботи із кластером ресурсів.

Результатом даного розділу є математична модель задачі планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації із описом її вхідних даних для аналізу.

3.1 Аналіз проблеми

Задача планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації полягає в знаходженні найменшої кількості хостів, тобто, віртуальних машин або фізичних машин, заздалегідь відомої ємності для розміщення фіксованої кількості контейнерів, кожен з яких має обмеження за ресурсами процесора та оперативної пам’яті.

В рамках Kubernetes задача планування ресурсної ємності формується як знаходження найменшої кількості вузлів кластера для розміщення вказаного набору под.

Дана задача є схожою до задачі консолідації віртуальних машин у ЦОД [14, 15, 16], проте важливою відмінністю є науково практичне значення, а саме, робота із контейнерним навантаженням та розподіленням його у кластері ресурсів, а не із віртуальними машинами та їх розподіленням у ЦОД. Тобто поставлена задача є

відмінною від задачі консолідації віртуальних машин, адже мається на увазі використання кардинально іншої технології (як було показано у розділі 1), та має істотно інші ресурсні та логічні обмеження.

У загальному сенсі, задача планування ресурсної ємності є проблемою, пов’язаною із пакуванням контейнерів (Bin packing) [17].

Оскільки у задачі планування ресурсної ємності, необхідно брати до уваги декілька напрямків, таких як процесорний час та кількість оперативної пам’яті, то задачу можна розглядати як задачу багатовимірного пакування (Multi-dimensional Bin packing). Проте, у задачі багатовимірного пакування, ресурси не виділені під окремий об’єкт, а можуть використовуватися одночасно, як показано на рисунку 3.1.

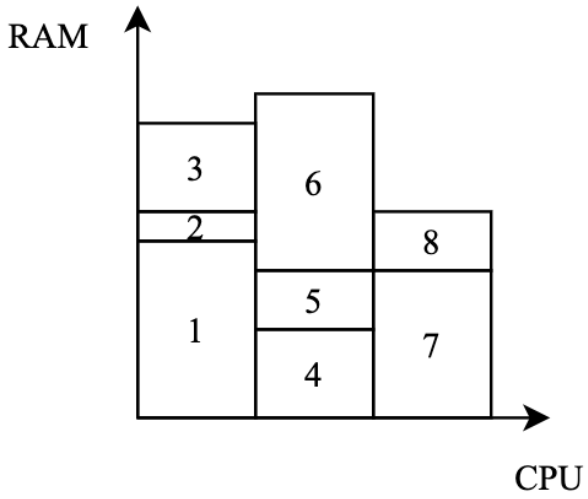


Рисунок 3.1 – Приклад задачі багатовимірного пакування

Як видно з рисунку, усі елементи використовують спільні ресурси. Наприклад, блок 1 використовує деяку частину RAM із блоками 4, 5, 6, 7, 8 та частину CPU із блоками 2, 3.

Таке використання ресурсів неможливе у контексті використання обчислювальних ресурсів, адже, якщо один блок вже зайняв ресурси, то вони мають належати тільки йому. Якщо ресурсна ємність складається із 4 Гб RAM та 2 vCPU, та блок 1 займає 1.5 Гб RAM та 0.3 CPU, то для інших ресурсів доступно лише 2.5 Гб RAM та 1.7 CPU.

У задачі планування ресурсної ємності усі блоки мають тільки виділені йому ресурси, і не використовують спільні, як показано на рисунку 3.2.

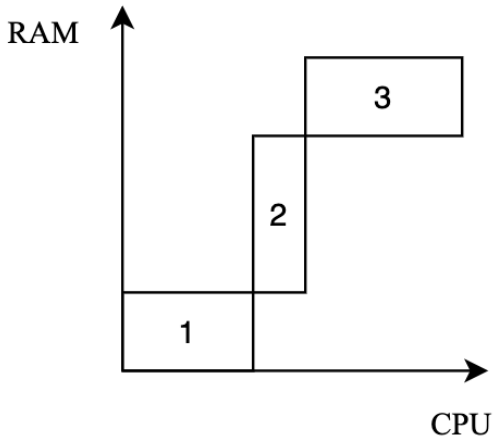


Рисунок 3.2 – Використання ресурсів у задачі планування ресурсної ємності

Формування задачі для Kubernetes є достатньо схожим, проте із використанням властивих для Kubernetes та контейнерного навантаження абстракцій. Абстрактне зображення задачі планування ресурсної ємності для Kubernetes зображено на рисунку 3.3.

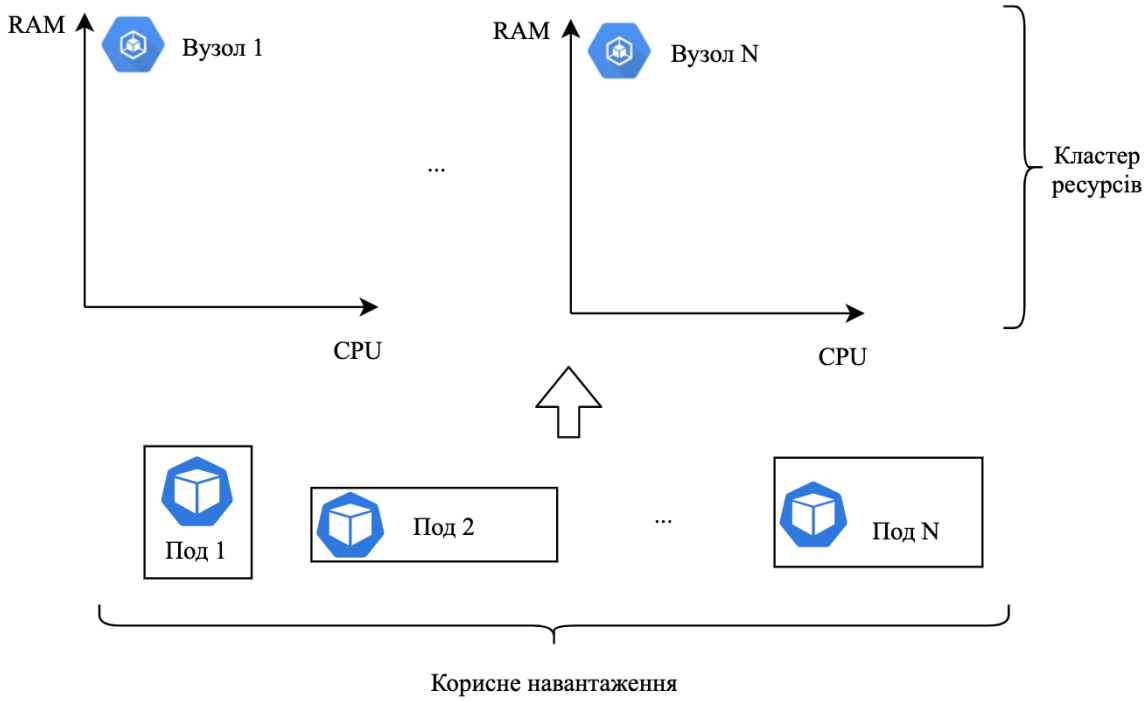


Рисунок 3.3 – Задача планування ресурсної ємності для Kubernetes

Проте, як було вказано в розділі 2, Kubernetes має абстракцію, яка об’єдную вузли із однаковими властивостями та ресурсною ємність у абстракцію, названу пулом вузлів.

У даному контексті, задача розширюється і доповнюється додатковим фактором, що розмірів вузлів у кластері не однаковий, а може мати різні варіації. Зазвичай, не використовуються більше 1-3 пулів вузлів, тому у даній роботі логіку вибору буде обмежено трьома типами вузлів.

Тобто, може існувати три типи вузлів, та кожен тип вузла має свою ресурсну ємність. Всередині пулу кількість вузлів є необмеженою. Візуалізацію задачі із використанням пулу вузлів наведено на рисунку 3.4.

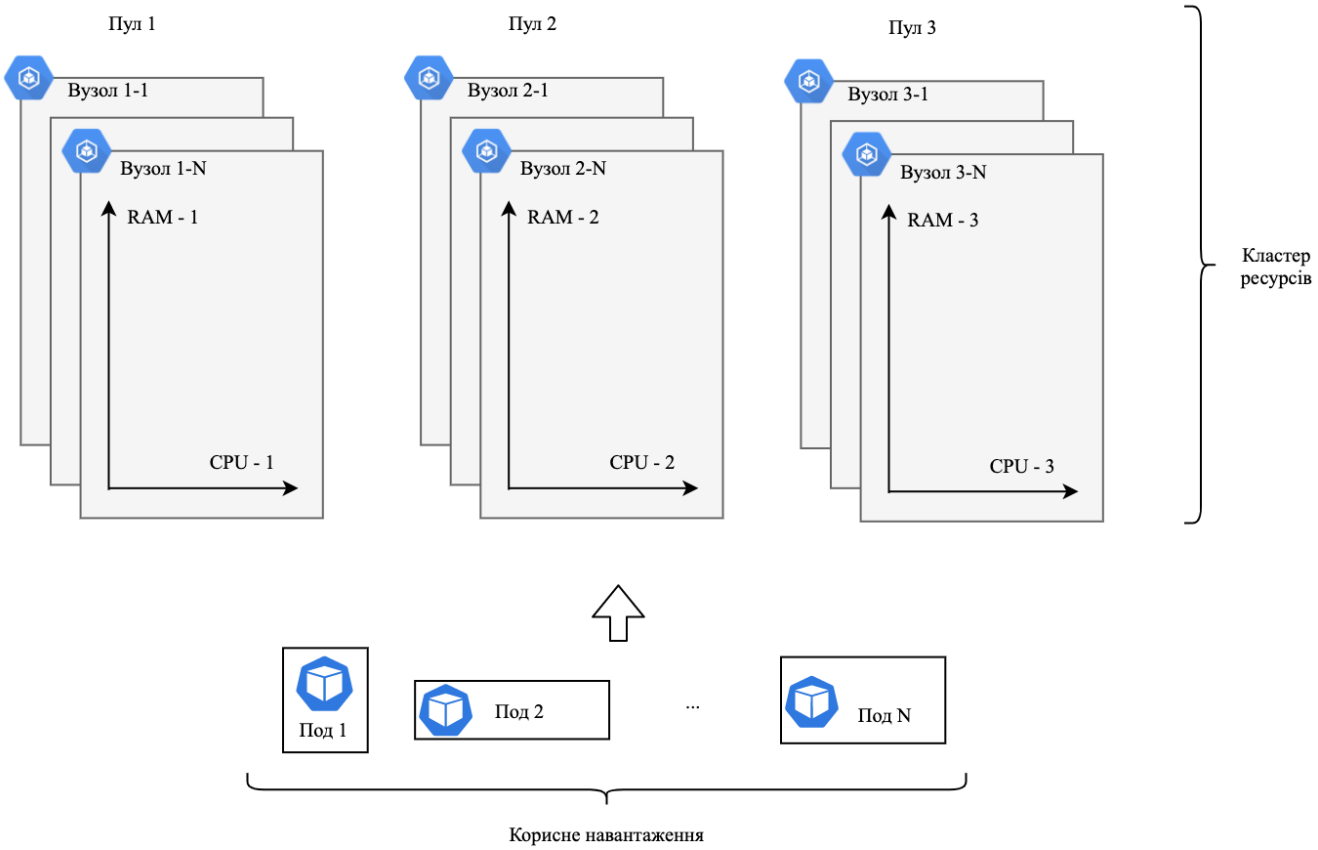


Рисунок 3.4 – Задача планування ресурсної ємності із використанням пулу вузлів

Отже, із прийняттям до уваги усіх особливостей та обмежень, задача планування ресурсної ємності формується як знаходження найменшої кількості

вузлів кластера у заданих пулах вузлів для розміщення вказаного набору под різного розміру.

Задача пакування контейнерів є NP-складною [18], тобто, не має поліноміального рішення. Для задач із великою кількість вхідних даних, час виконання алгоритму є дуже великим, що, зазвичай, не відповідає функціональним та нефункціональним вимогам, поставленим до продукту. Зазвичай, для вирішення задач такого класу використовуються різні евристичні алгоритми.

3.2 Офлайн та онлайн планування ресурсної ємності

Планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації поділити на дві основні логічні частини:

- офлайн – коли заздалегідь відоме усе контейнерне навантаження, яке має бути розгорнуто на вузлах кластера. Задача такого планування формується як задача пакування контейнерів при початковому стані кластера, коли ще немає інших контейнерів кластера;
- онлайн – коли усе контейнерне навантаження заздалегідь невідоме та, ймовірно, кластер вже має деяку кількість вже розгорнутих на ньому контейнерів.

Задача офлайн планування ресурсної ємності є основним фокусом даної роботи, проте, буде розглянуто і особливості онлайн пакування.

У задачі офлайн планування все сходиться до вирішення задачі пакування контейнерів в умовах однакового розміру контейнерів (у випадку з Kubernetes – вузлів кластера), в розрізі наявності декількох пулів вузлів. Усі вузли в рамках одного пулу мають однаковий розмір.

Проте, у задачі онлайн пакування, в кластері вже є деяке навантаження, тому кожен вузол може знаходитися у двох станах:

- порожній – на даному вузлі немає ні одного пода. Всі його ресурси можуть бути використані;

– має навантаження – на даному вузлі вже є деякі поди, тому його максимальна ємність не дорівнює ємності, яка доступна для розгортання нового навантаження.

Тобто, у рамках одного пулу вузлів існує різна ресурсна ємність, доступна для використання. Тоді задача сходиться до задачі багатовимірного пакування із контейнерами різного розміру. В рамках одного пулу можуть знаходитися вузли різного залишкового розміру, проте, нові вузли у пулі будуть мати повну місткість. Схематично задачу із онлайн планування показано на рисунку 3.5. На рисунку 3.5 N – кількість вже використаних вузлів із пулу, а N + 1 – новий, не використаний вузол.

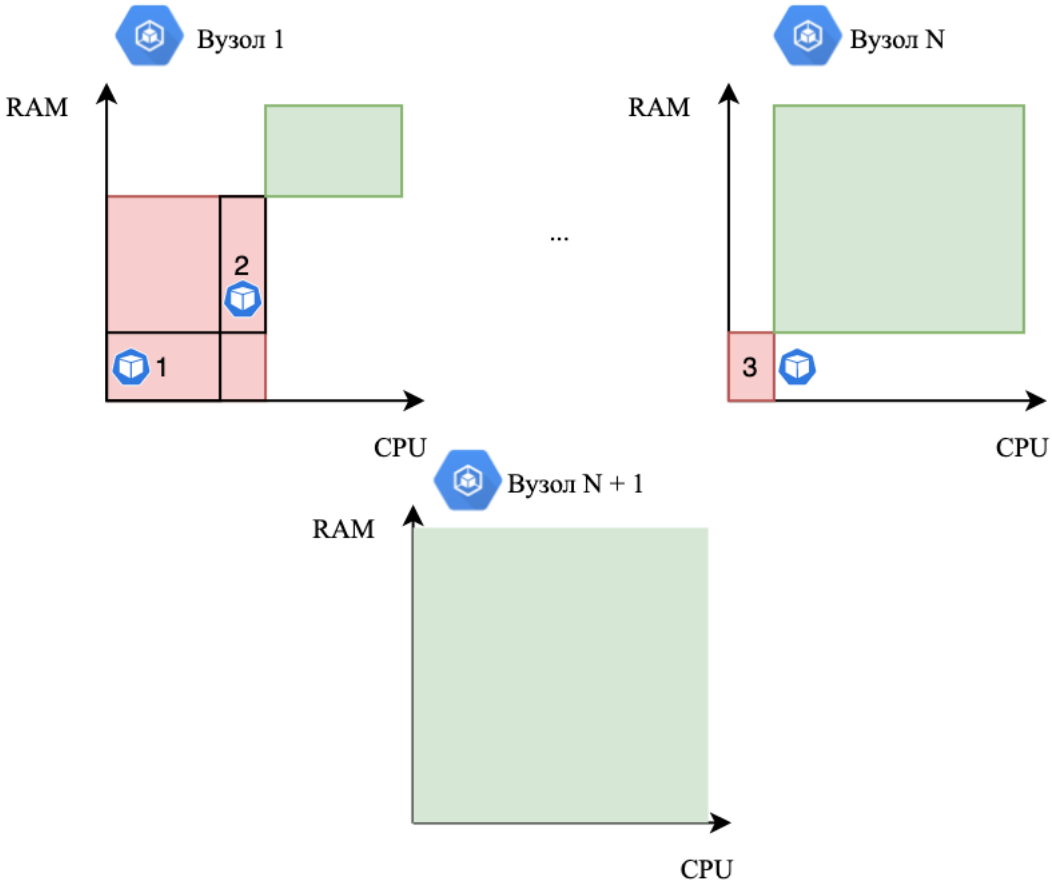


Рисунок 3.5 – Схема задачі онлайн планування

На даному рисунку червоною областю позначено ресурси вузлів, які вже зайнято відповідними подами, тобто, ті ресурси, які не можуть бути використані для планування.

Зеленою областю вказано вільні ресурси, які можна використовувати для планування додаткового навантаження. Як видно, зелені області мають різні розміри, що свідчить про те, що під час онлайн планування, наявні ресурси кожного вузла, можуть відрізнятися від максимальних.

3.3 Математична модель

Математична модель – це опис системи з використанням математичних понять і мови. Процес розробки математичної моделі називається математичним моделюванням. Математичні моделі використовуються в природничих науках і інженерних дисциплінах, а також в нефізичних системах, таких як соціальні науки (наприклад, економіка, психологія, соціологія, політологія).

Математичні моделі можуть приймати різні форми, включаючи динамічні системи, статистичні моделі, диференціальні рівняння або теоретико-ігрові моделі. Ці та інші типи моделей можуть перетинатися, при цьому кожна модель може включати в себе безліч абстрактних структур. Загалом, математичні моделі можуть включати логічні моделі. У багатьох випадках якість наукової області залежить від того, наскільки добре математичні моделі, розроблені з теоретичного боку, узгоджуються з результатами повторюваних експериментів. Невідповідність між теоретичними математичними моделями і експериментальними вимірами часто призводить до важливих досягнень у міру розробки досконаліших теорій.

Надалі описано математичну модель планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації.

Нехай, $S = S_1, S_2, \dots, S_m$ набір вузлів у кластері, та кожен вузол має розмір c. Нехай, $w_1, w_2, \dots, w_n$ – набір розмірів под, які мають бути поміщені на вузлах кластера.

Задача полягає в тому, щоб призначити кожному поду вузол, так, щоб, сумарна кількість ваг усіх подів на окремому вузлу не перевищувала ємність c та кількість використаних вузлів є мінімальною. Тобто, знайти таке число z вузлів, щоб:

$$\min z = \sum_{i=1}^{n} y_i \tag{3.1}$$

Де $z \geq 1$,

$\sum_{j=1}^{n} w_j x_{ij} \leq c y_i, \forall i \in (1,2,\dots,n)$

$\sum_{i=1}^{n} x_{ij} = 1, \forall j \in (1,2,\dots,n)$,

$y_i \in (0,1), \forall i \in (1,2,\dots,n)$,

$x_{ij} \in (0,1), \forall i \in (1,2,\dots,n), \forall j \in (1,2,\dots n)$,

тут $y_i = 0$, якщо вузол i не використано, інакше $y_i = 1$, та $x_{ij} = 1$, якщо под j додано до вузла i, інакше $x_{ij} = 0$.

У формулі 3.1 наведено рівняння мінімізації кількості використаних вузлів для розташування вказаної кількості под. Таку математичну модель можна використовувати, якщо поди і вузли мають лише одиницю виміру, наприклад, тільки CPU, або ж тільки RAM.

Проте, більш практично використовувати не тільки один вимір, а декілька, тобто, як поди, так і вузли вимірювати у двох напрямках: CPU та RAM. Тоді, ресурси под та вузлів кластера формуються так, як вказано у рівняннях 3.2 та 3.3 відповідно.

$$w_i = \left(w_i^1, w_i^2, \dots, w_i^d\right) \tag{3.2}$$

де $d \geq 1, i \in (1, 2, \dots, n)$.

$$c = (c^1, c^2, \dots, c^d) \tag{3.3}$$

де $d \geq 1$.

Тоді, маємо нерівність, вказану у формулі 3.3 для багатовимірного планування та кількості вимірів $D = \{1, 2, \dots d\}$:

$$\sum_{j=1}^{n} w_j^d x_{ij} \leq c^d y_i \tag{3.4}$$

де $d \in D$,

$\forall i \in (1,2,\dots,n)$

Рівняння 3.1 та нерівність 3.4 визначають задачу планування ресурсної ємності кластера ресурсів для заданого навантаження при умові, що всі вузли мають ємність c, як вказано у формулі 3.3.

Якщо ж ресурси кожного вузла не є однаковими, як для задачі онлайн планування, то ємність вузла має бути описана так, як показано у формулі 3.5.

$$c_k = \left(c_k^1,\ c_k^2, \dots, c_k^d\right) \tag{3.5}$$

де $d \geq 1, k \geq 1$.

Тоді, якщо вузол було використано, то для усіх подів, які розташовані на вузлах вираз 3.6 є справедливим.

$$\sum_{i=1}^{B} w_j^d x_{ij} \leq c_i^d \tag{3.6}$$

де B – кількість використаних вузлів,

$d \geq 1$,

$\sum_{\mathrm{i=1}}^{\mathrm{B}} \mathrm{x_{ij}} = 1, \forall \mathrm{j} \in (1,2, \dots, \mathrm{n})$,

$\mathrm{x_{ij}} \in (0,1), \forall \mathrm{i} \in (1,2, \dots, \mathrm{B}), \forall \mathrm{j} \in (1,2, \dots n)$,

та $x_{ij} = 1$, якщо под j додано до вузла i, інакше $x_{ij} = 0$.

Для виконання деяких алгоритмічних операцій можуть знадобитися перетворення багатовимірного вектора у скаляр. Для деякого пода w_i, це буде виконуватися за допомогою формул 3.7 та 3.8 в залежності від потреби.

$$wprod(w_i) = \prod_{j \leq d} a_j w_i^j \tag{3.7}$$

$$wsum(w_i) = \sum_{j \leq d} a_j w_i^j \tag{3.8}$$

Вектор $\mathrm{a} = a_1, a_2, \dots, a_d$ – вектор вибору, який дозволяє за допомогою коефіцієнтів балансувати важливість кожного з заданих розрізів. Наприклад, можливо надати більшу частину пріоритету до CPU, аніж до RAM, якщо в кластері вузьким місцем є саме CPU, а RAM не є критичним ресурсом.

Цим самим, кінцевий користувач алгоритму планування ресурсної ємності може задати пріоритети щодо того чи іншого ресурсу, або ж, навіть нівелювати один з ресурсів та використовувати для підрахунків лише один.

3.4 Висновки до розділу

У даному розділі було проаналізована проблему планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації. Задачу планування окреслено як задачу багатовимірного пакування контейнерів.

Для задачі планування можуть використовуватися, як вузли одного типу, тобто, однієї ресурсної ємності, так і декількох типів – із різник пулів вузлів.

Розглянуті основні властивості планування у режимі офлайн та онлайн. У режимі офлайн навантаження є заздалегідь відомим та кластер знаходиться у початковому стані, тобто немає поточного навантаження. При роботі у режимі онлайн кластер вже може мати деяке навантаження, та майбутнє навантаження заздалегідь невідоме.

Також, побудовано математичну модель задачі, яку буде використано при побудові алгоритмів та їх програмній реалізації у наступних розділах.

4 РОЗРОБКА АЛГОРИТМУ ПЛАНУВАННЯ РЕСУРСНОЇ ЄМНОСТІ

У даному розділі на базі аналізу проблеми та математичної моделі, описаної у розділі 3, буде запропоновано алгоритм планування ресурсної ємності для заданого навантаження.

Метою даного розділу є побудова алгоритму, його програмна реалізація та проведення експериментів із різними вхідними даними для запропонованих алгоритмів.

Наступні алгоритми мають бути спроектовані:

– алгоритм планування ресурсної ємності при порожньому кластері, тобто, офлайн планування;
– алгоритм планування ресурсної ємності при кластері, в якому вже є деяке корисне навантаження, тобто, онлайн планування;
– алгоритм розподілення пода до вузла із масиву пулів вузлів.

У даному розділі наведено декілька алгоритмів для планування, як у порожньому кластері, так і в кластері із навантаженням, порівняно експерименти із даними алгоритмами, порівняно результати.

4.1 Алгоритм офлайн планування

Алгоритм офлайн планування має на меті упакувати задані кінцевим користувачем поди різного розміру кожний у найменшу кількість вузлів кластера із одного його пулу вузлів.

Для вирішення цієї задачі буде використано алгоритми Next Fit, First Fit, Best Fit, Worst Fit, та їх модифікації із зміною порядку задання под на вхід, наприклад, Best Fit Decreasing. Також, запропоновано власний алгоритм для вирішення поставленої задачі.

Для кожного алгоритму буде наведено його програмну реалізацію з її детальним поясненням.

4.1.1 Next Fit

Next Fit – це алгоритм з обмеженим простором, в якому в будь-який момент може бути відкртитий для наповнення тільки одни вузол.

Алгоритм працює наступним чином: поди розглядаються у тому порядку, як вони були надані; якщо под може бути поміщено у поточний вузол, то він поміщається до нього. Якщо ж под не поміщаєтьс, поточний вузол закривається, відкривається новий та поміщається всередину нового.

Програмний код для алгоритму Next Fit для пакування кластера Kubernetes наведено нижче. Повний код застосунку наведено у додатку Б.

```
def estimate(
    self,
    initial_nodes: List[Node],
    node_capacity: Capacity,
    pods: Sequence[Pod],
    vector_converter: VectorConverter
) -> EstimationResult:
    nodes = initial_nodes
    nodes_created = 0
    current_node = None

    for pod in pods:
        can_fit = False

        if current_node and current_node.can_fit(pod):
            can_fit = True
            current_node.add_pod(pod)

        if not can_fit:
            new_node = Node.generate_with_capacity(node_capacity)
            new_node.add_pod(pod)
            nodes.append(new_node)

            current_node = new_node
            nodes_created += 1
```

```
    return EstimationResult(nodes, nodes_created)
```

Для кожного поду зі списку проводяться наступні операції:

– обчислюється, чи може бути поміщений поточний под у поточний вузол;
– якщо може бути поміщений, то переходимо до наступного поду;
– якщо ні, то відкривається новий вузол і под поміщується у нього. Новий вузол стає поточним.

4.1.2 First Fit

First Fit – це алгоритм, який працює так, що поміщує под у перший вузол, який його може вмістити.

Поди розглядаються у тому порядку, як вони були надані; для всіх наявних вузлів, по черзі, алгоритм намагається розподілити под на вузол. Якщо вдалося розподілити, то алгоритм переходить до наступного пода. Якщо вже пройдено усі вузли, та не вдалося розподілити вузол, то створюється новий вузол, і под поміщається у нього.

Програмний код для алгоритму First Fit для пакування кластера Kubernetes наведено нижче.

```
def estimate(
    self,
    initial_nodes: List[Node],
    node_capacity: Capacity,
    pods: Sequence[Pod],
    vector_converter: VectorConverter
) -> EstimationResult:
    nodes = initial_nodes
    nodes_created = 0

    for pod in pods:
        existing_node_used = False

        for node in nodes:
            if node.can_fit(pod):
                existing_node_used = True
```

```
            node.add_pod(pod)

        if not existing_node_used:
            new_node = Node.generate_with_capacity(node_capacity)
            new_node.add_pod(pod)
            nodes.append(new_node)

            nodes_created += 1

    return EstimationResult(nodes, nodes_created)
```

Для кожного поду зі списку проводяться наступні операції:

- серед усіх вузлів знайти перший вузол, який може помістити под;
- якщо вдалося знайти вузол, то помістити под у знайдений вузол та перейти до іншого поду;
- якщо вузол не знайдено, то створити новий вузол та помістити под у нього. Новий вузол помістити у список усіх вузлів.

4.1.3 Best Fit

Best Fit – алгоритм, схожий до First Fit, проте поміщує под у той вузол що, на поточний час має найбільшу завантаженість.

Поди розглядаються у тому порядку, як вони були надані; Для кожного наявного вузла, обчислюється, чи може вузол помістити новий под. Якщо може помістити, то обчислити кількість вільних ресурсів і знаходиться той вузол, що може помістити под, та має найменшу кількість вільних ресурсів.

Якщо не знайдено вузлів, що можуть помістити под, то відкривається новий вузол.

Програмну реалізацію Best Fit алгоритму для пакування Kubernetes кластера надано нижче.

```
def estimate(
    self,
    initial_nodes: List[Node],
    node_capacity: Capacity,
    pods: Sequence[Pod],
```

```
    vector_converter: VectorConverter
) -> EstimationResult:
    nodes = initial_nodes
    nodes_created = 0

    for pod in pods:
        existing_node_used = False
        best_node_index = -1
        min_score = 0

        for idx, node in enumerate(nodes):
            if not node.can_fit(pod):
                continue

            scalar_node = vector_converter.convert(capacity=node.remaining_capacity)
            score = scalar_node

            if best_node_index == -1:
                min_score = score
                best_node_index = idx
            else:
                if score < min_score:
                    best_node_index = idx
                    min_score = score

        if best_node_index != -1:
            existing_node_used = True
            nodes[best_node_index].add_pod(pod)

        if not existing_node_used:
            new_node = Node.generate_with_capacity(node_capacity)
            new_node.add_pod(pod)
            nodes.append(new_node)

            nodes_created += 1

    return EstimationResult(nodes, nodes_created)
```

Для кожного поду зі списку проводяться наступні операції:

- серед усіх вузли, що можуть вмістити заданий под;
- серед знайдених вузлів знайти той, що має найменшу кількість вільних ресурсів;
- помістити под у той вузол, що має найменшу кількість вільних ресурсів;
- Якщо не знайдено вузлів, то відкрити новий вузол і додати под до нього. Новий вузол додати до списку вузлів.

Особливістю розробки даного алгоритму є перехід між багатовимірним значенням: кількість CPU та оперативної пам’яті у скалярне значення. Таких перехід робиться за допомогою формул 3.7 та 3.8.

За допомогою вказаних формул, багатовимірне значення перетворюється на скалярне, що дозволяє уніфіковано порівнювати ресурси, сортувати поди за розміром тощо.

4.1.4 Worst fit

Worst Fit – алгоритм, схожий до Best Fit, проте поміщує под у той вузол що, на поточний час має найменшу завантаженість.

Поди розглядаються у тому порядку, як вони були надані; Для кожного наявного вузла, обчислюється, чи може вузол помістити новий под. Якщо може помістити, то обчислюється кількість вільних ресурсів і знаходиться той вузол, що може помістити под, та має найбільшу кількість вільних ресурсів.

Якщо не знайдено вузлів, що можуть помістити под, то відкривається новий вузол. Програмну реалізацію Worst Fit алгоритму для пакування Kubernetes кластера надано нижче.

```
def estimate(
    self,
    initial_nodes: List[Node],
    node_capacity: Capacity,
    pods: Sequence[Pod],
    vector_converter: VectorConverter
) -> EstimationResult:
```

```
    nodes = initial_nodes
    nodes_created = 0

    for pod in pods:
        existing_node_used = False
        best_node_index = -1
        max_score = 0

        for idx, node in enumerate(nodes):
            if not node.can_fit(pod):
                continue

            scalar_node = vector_converter.convert(capacity=node.remaining_capacity)
            score = scalar_node

            if best_node_index == -1:
                max_score = score
                best_node_index = idx
            else:
                if score > max_score:
                    best_node_index = idx
                    max_score = score

        if best_node_index != -1:
            existing_node_used = True
            nodes[best_node_index].add_pod(pod)

        if not existing_node_used:
            new_node = Node.generate_with_capacity(node_capacity)
            new_node.add_pod(pod)
            nodes.append(new_node)

            nodes_created += 1

    return EstimationResult(nodes, nodes_created)
```

Для кожного поду зі списку проводяться наступні операції:

– серед усіх вузли, що можуть вмістити заданий под;

- серед знайдених вузлів знайти той, що має найменшу кількість вільних ресурсів;
- помістити под у той вузол, що має найбільшу кількість вільних ресурсів;
- Якщо не знайдено вузлів, то відкрити новий вузол і додати под до нього. Новий вузол додати до списку вузлів.

Для Worst Fit, як і для Best Fit використовується трансформація багатовимірного значення у скалярне для порівняння залишкової ресурсної ємності в уніфікованому вигляді.

4.1.5 Гібридний алгоритм

Одним з основних недоліків алгоритмів Best Fit та Worst Fit є те, що, використовується вузол, які має граничні для алгоритму значення: або найменшу кількість вільних ресурсів, або ж найбільшу кількість. Така поведінка в деяких випадках може призводити до неоптимальної комплектації контейнерів.

Для вирішення даної проблеми пропонується використання гібридної версії даних алгоритмів: замість використання граничного значення – використовувати значення, яке знаходиться до граничного, тобто, передостаннє.

Наприклад, у Best Fit використовувати не той вузол, що має найменше ресурсів, а той що передує цьому вузлу, тобто, той, що має друге нейменше значення вільних ресурсів. Для Worst Fit аналогічно – використовувати передостанній вузол, що має друге найбільше значення вільних ресурсів.

Програмна реазілація гібридного Best Fit алгоритму наведена нижче.

```
def estimate(
    self,
    initial_nodes: List[Node],
    node_capacity: Capacity,
    pods: Sequence[Pod],
    vector_converter: VectorConverter
) -> EstimationResult:
    nodes = initial_nodes
    nodes_created = 0
```

```
for pod in pods:
    existing_node_used = False
    best_node_index = -1
    prev_node_index = -1
    min_score = 0

    for idx, node in enumerate(nodes):
        if not node.can_fit(pod):
            continue

        scalar_node = vector_converter.convert(capacity=node.remaining_capacity)
        score = scalar_node

        if best_node_index == -1:
            min_score = score
            best_node_index = idx
            prev_node_index = idx
        else:
            if score < min_score:
                prev_node_index = best_node_index
                best_node_index = idx
                min_score = score

    if best_node_index != -1:
        existing_node_used = True
        if nodes[prev_node_index].can_fit(pod):
            nodes[prev_node_index].add_pod(pod)
        else:
            nodes[best_node_index].add_pod(pod)

    if not existing_node_used:
        new_node = Node.generate_with_capacity(node_capacity)
        new_node.add_pod(pod)
        nodes.append(new_node)

        nodes_created += 1

return EstimationResult(nodes, nodes_created)
```

Для кожного пода виконуються наступні дії:

– у списку знаходиться вузол, який має найменше ресурсів, та той, що другий за найменшою кількістю ресурсів;
– под додається у другий за найменшою кількістю ресурсів вузол, якщо підходящий вузол тільки один, то под додається до нього.
– якщо підходящих вузлів не знайдено, то відкривається новий вузол і под додається до нього. Новий вузол додається до списку вузлів;

Для модифікації Worst Fit, необхідно використовувати другий за найбільшою кількістю вільних ресурсів. Всі інші кроки залишити без змін.

4.1.6 Зміна порядку под для розміщення

В усіх вищенаведених алгоритмах використовується порядок под такий, як його було було надано користувачем, тобто, без змін. Цей порядок може бути як випадковий, так і упорядкований за деяким принципом.

Надані алгоритми можна модифікувати так, щоб вони працювали із впорядкованим набором под. Порядок под можна визначити шляхом сортування їх за скалярним значенням їх необхідних ресурсів. Ресурси поду можна перевести до скалярних значень за допомогою формул 3.7 та 3.8.

Можливо використовувати наступні варіації алгоритмів зі зміною порядку розміщення по:

– без зміни порядку под;
– сортувати за спаданням скалярного значення ресурсів поду;
– сортувати за зростанням скалярного значення ресурсів поду.

Програмну реалізацію сортування наведено нижче.

```
def sort_key(px):
   scalar_px = vector_converter.convert(px.requests)
   return scalar_px

pods_actual = pods
```

```
pods_decreasing = sorted(pods, key=lambda x: sort_key(x), reverse=True)
pods_increasing =  sorted(pods, key=lambda x: sort_key(x), reverse=False)
```

4.1.7 Загальний алгоритм планування

У попередніх підпунктах були розглянуті такі алгоритми, як Next Fit, First Fit, Best Fit, Worst Fit, запропоновано власний гібридний варіант алгоритму, а також розглянуто варіації алгоритмів із зміною порядку обробки подів, які необхідно розгорнути на кластері: за спаданням та зростанням ресурсів, а також без зміни природнього порядку.

Кожен алгоритм, при певному наборі даних може показати кращі, або ж гірші результати, адже ці алгоритми є евристичними та не гарантують повністю точного рішення, а лише гарантують, що рішення буде приближене до правильного.

Алгоритм планування ресурсної ємності для заданого навантаження має містити наступні етапи:

- обробка вхідних даних від користувача. На даному етапі необхідно отримати усі поди, які мають бути розташовані у кластері ресурсів Kubernetes. Важливими даними є ресурси, необхідні для поду, адже, саме на базі цих даних буде формуватися кластер;
- вибір правильного алгоритму планування. Для всіх заданих под необхідно обчислити кількість вузлів, яку необхідно додати до кластера, та обрати той алгоритм, який вимагає найменшої кількості вузлів;
- для кожного пода зі списку знайти відповідний йому вузол;
- додати до кластера необхідну кількість вузлів;
- розташувати усі поди на вузлах кластера.

При виборі правильного алгоритму планування мають використовуватися наступні алгоритми:

- Next Fit, Next Decreasing, Next Fit Increasing;
- First Fit, First Decreasing, First Fit Increasing;
- Best Fit, Best Decreasing, Best Fit Increasing;

– Worst Fit, Worst Decreasing, Worst Fit Increasing;
– Запропонований гібридний алгоритм.

Для кожного із алгоритмів – обчислити кількість візлів, яку необхідно використати для розташування усіх заданих под.

Також, із алгоритму необхідно отримати розподілення який под має бути на якому вузлі. Надалі із розподілу, отриманого за допомогою алгоритму, розташувати поди на вузлах.

Розташування поду на конкретному вузлі може виконуватися за допомогою директиви nodeSelector [19] у Kubernetes. nodeSelector – це найпростіша рекомендована форма обмеження вибору вузла. nodeSelector – це поле PodSpec. Він визначає карту пар ключ-значення. Щоб под міг працювати на вузлі, вузол повинен мати кожну із зазначених пар ключ-значення як мітки (він може також мати додаткові мітки). Найбільш поширене використання - одна пара ключ-значення. Наприклад, на рисунку 4.1 наведена конфігурація, яка використовує nodeSelector.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

Рисунок 4.1 – Приклад використання nodeSelector

Тут використовується nodeSelector disktype: ssd. Це значить, що под буде розгорнуто лише на вузлах, які мають мітку disktype: ssd. У випадку із явним розташуванням поду на вузлах та унікальній комбінації вузол: под, вузлам буде надано унікальний id, щоб точно ідентифікувати вузол у кластері, та розгортати саме так, як видає алгоритм.

Діаграму активності для даного алгоритму наведено у додатку Г. На діаграмі активності використано три основні групи активносетей:

– підготовка алгоритму. Тут зчитуються дані про поди, які мають бути розташовані у кластері, валідуються вхідні дані. Також, зчитується та валідується цільовий розмір вузла;
– планування ресурсної ємності. На даному етапі аикористовуються всі запропоновані алгоритми із трьома можливими перестановками подів. Всі результати зберігаються. Для операції «циклу» використано директиву <<iterative>>;
– аналіз результатів. Аналізуються результати, та, якщо результати надали розподілення подів між вузлами, то, надалі, поди розташовуються на вузлах за заданим розподілом.

4.2 Проведення експериментів над алгоритмами

Для валідації запропонованих раніше алгоримтів створено тестову вибірку із випадкових значень запитів под та цільових ресурсів вузла. Алгоритми порівнюються за наступними критеріями:

– кількість використаних вузлів для розташування усіх заданих под;
– середня утилізація CPU на кожному вузлі;
– середня утилізація RAM на кожному вузлі.

Також, для перевірки якості алгоритму виконується порівняння його результатів із теоретичною найнижчою границею. Найнижча границя обраховується як максимальне значення серед наступних величин:

– сума усіх запитів под по CPU, розділена на CPU одного вузла;
– сума усіх запитів под по RAW, розділена на RAM одного вузла.

Це теоретична груба оцінка найнижчої границі алгоритму, створена для порівняння якості алгоритму.

Утилізація ресурсів обраховується як процентне відношення використаних ресурсів на вузлі до його максимальних.

Тестову вибірку для експериментів показано у таблиці 4.1.

Таблиця 4.1 – Тестова вибірка експерименту

Номер експерименту	Діапазон CPU для под	Діапазон RAM для под	Кількість под	Розмір вузла: CPU, RAM
1	[150, 550]	[128, 512]	1000	2000, 2048
2	[200, 500]	[256, 2048]	5000	4000, 16384
3	[100, 350]	[256, 712]	3000	1000, 2048

Відповідно до таблиця 4.1 буде згенеровано три експерименти. У колонці «Діапазон CPU для под» вказано у яких межах будуть випадково, для кожного пода, обрані значення необхідної кількості процесора (вимірюється у мілісекундах). У колонці «Діапазон RAM для под» вказано у яких межах будуть випадково, для кожного пода, обрані значення необхідної кількості RAM (вимірюється у мегабайтах). Кількість под вказує на розмір вибірки. Розмір вузла вказує кількість CPU та RAM для цільового розміру вузла.

Для кожного експерименту буде побудовано графіки:

– кількості використанних вузлів для кожного алгоритму. Менші значення, є кращими. На графіку буде відображено теоретичну нижню границю;
– утилізація CPU. Більші значення є кращими, максимальне значення – 100%. Вимірюється у відсотках;
– утилізація RAM. Більші значення є кращими, максимальне значення – 100%. Вимірюється у відсотках.

Результати експерименту 1 показано на рисунку 4.2. Аналізуючи результати можна дійти висновку, що найкраще показав себе алгоритми групи Hybrid Worst Fit, а саме: Hybrid Worst Fit Decreasing, Hybrid Worst Fit Increasing, та Hybrid Worst Fit As Is. Hybrid алгоритми було розроблено у підпункті 4.1.5.

Схожі результати показують алгоритми Best Fit та Worst Fit, проте, дещо гірші за гібридні алгоритми.

Утилізація процесора для вказаних алгоритмів вища за 95%, а утилізація оперативної пам’яті – вища за 85%, що є достатньо хорошим результатом

-

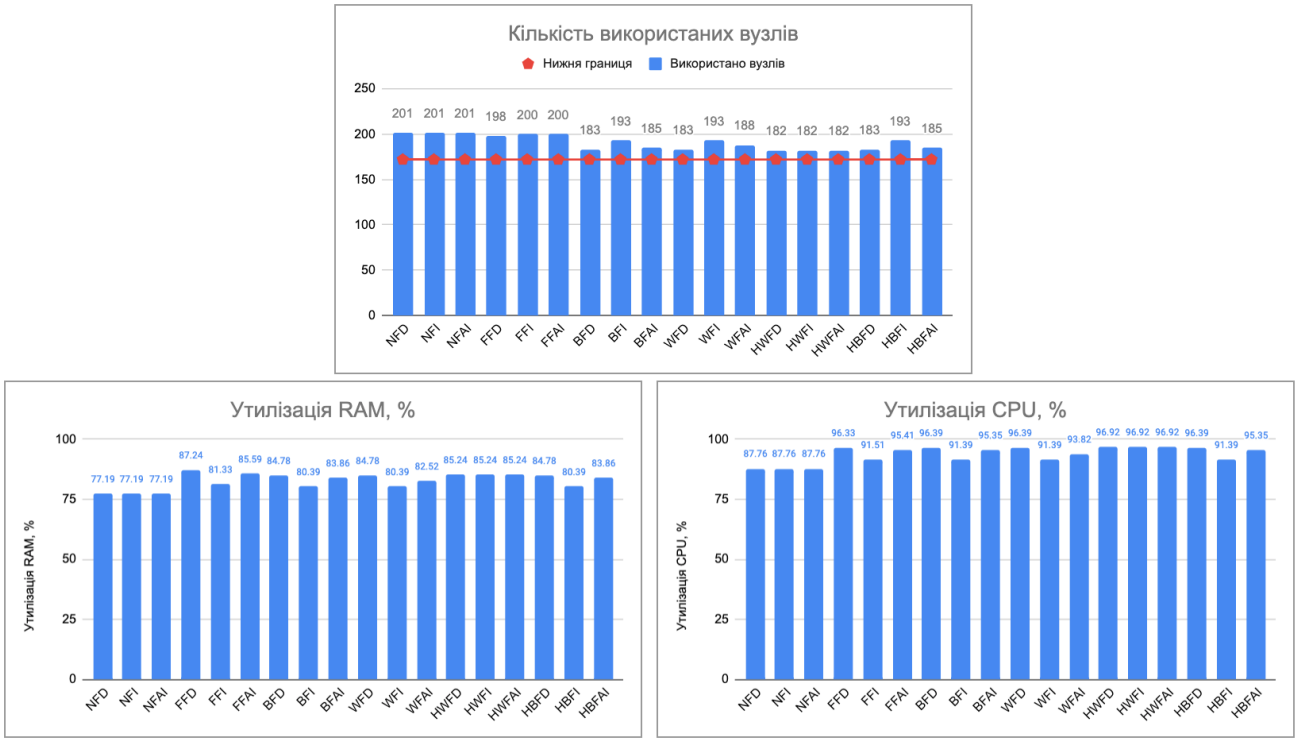


Рисунок 4.2 – Результати експерименту 1

Результати експерименту 2 показано на рисунку 4.3.

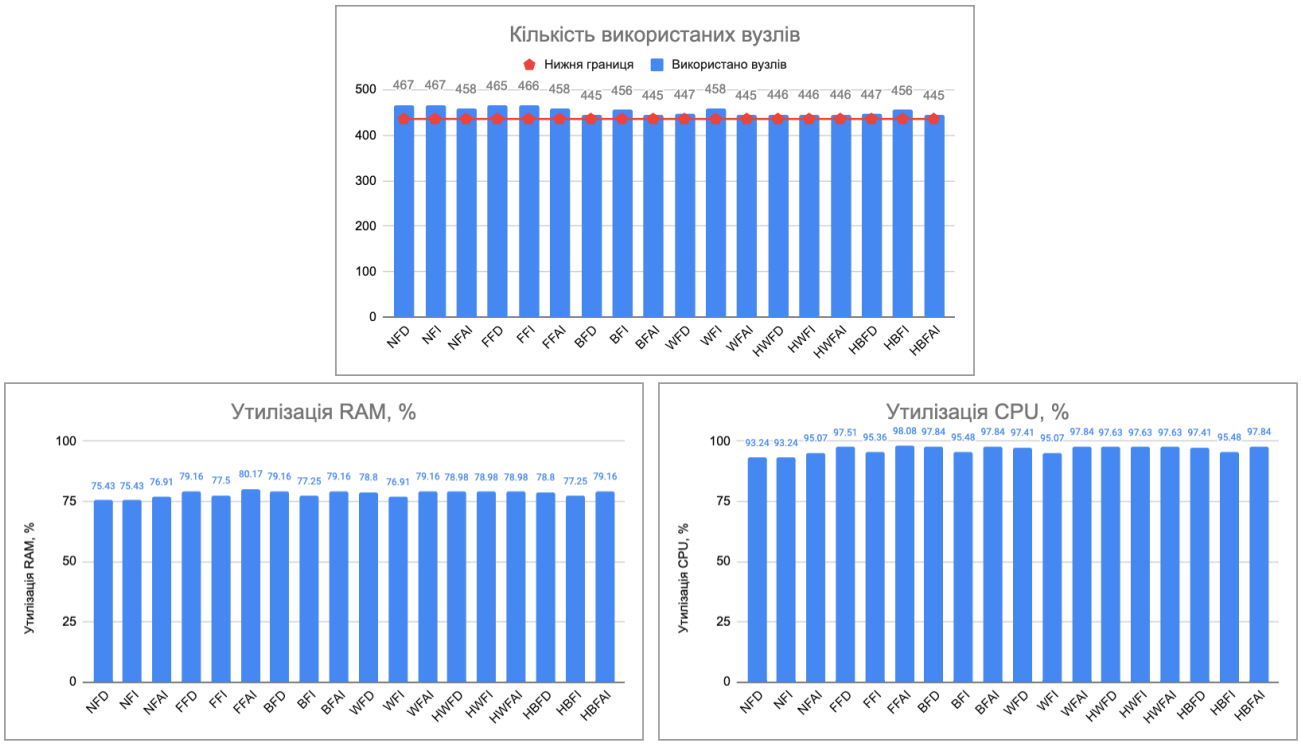


Рисунок 4.3 – Результати експерименту 2

Відповідно до графіків, зображених на рисунку 4.2, найкраще себе проявляють алгоритми Best Fit Decreasing, Best Fit As Is, Worst Fit As Is, Hybrid Best Fit As Is, тобто, група алгоритмів Best Fit, Worst Fit та Hybrid Fit, як і під час експерименту 1.

Утилізація ресурсів для вказаних алгоритмів хороша: утилілізація процесора свище 97%, а утилізація оперативної пам’яті свище 78%. Такий відсоток утилізації оперативної пам’яті пов’язано із специфікою експерименту, адже домінуючим ресурсом був процесорний час.

Результати експерименту 3 показано на рисунку 4.4.

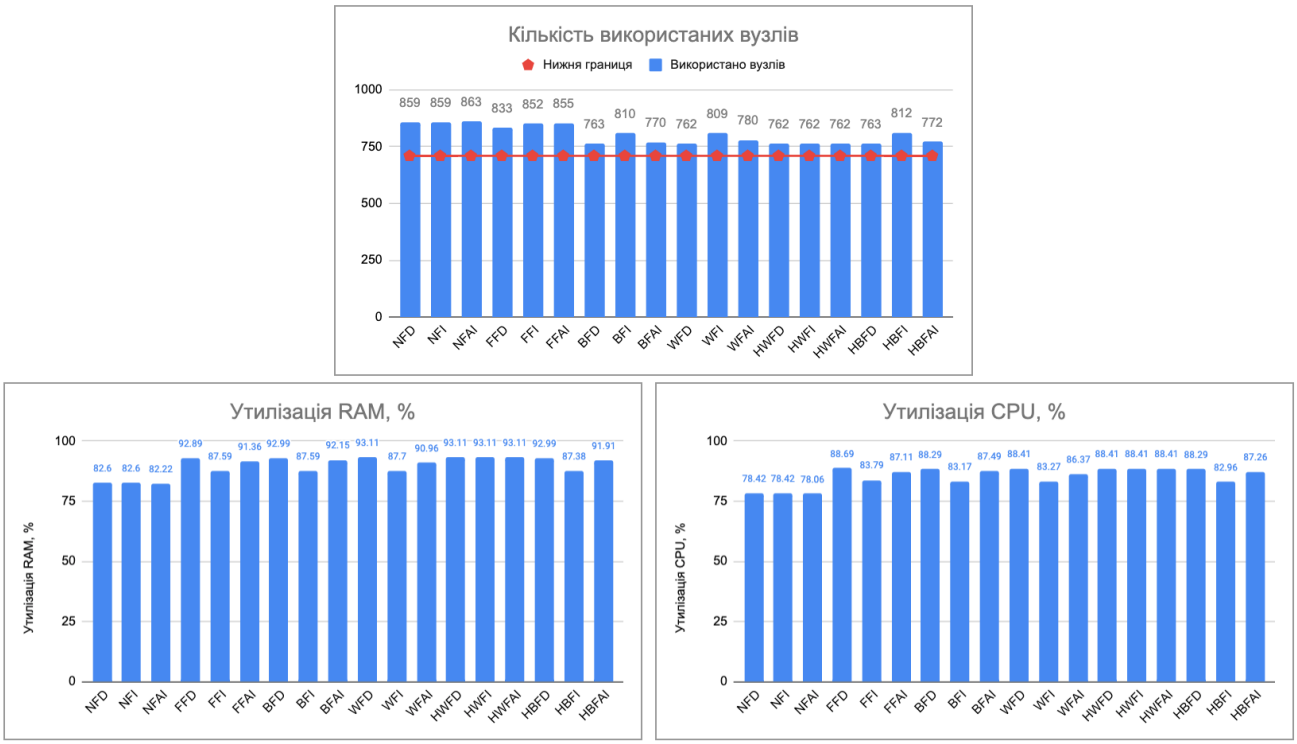


Рисунок 4.4 – Результати експерименту 3

Відповідно до графіків, показаних на рисунку 3, видно, що лідерами серед алгоритмів залишаються алгоритми із класів Best Fit, Worst Fit, та гібридних алгоритмів.

Отже, можна дійти висновку, що в більшості випадків найкраще себе проявляють алгоритми із класів Best Fit, Worst Fit, та гібридних алгоритмів. Проте,

вони є евристичними, та на деякому наборі даних можуть давати погані результати. Алгоритми з інших класів, таких як Next Fit, First Fit показують результати близькі до вказаних найкращих алгоритмів.

Також, аналізуючи результати, можна дійти висновку, що сортування елементів позитивно впливає на результати.

Тому, при вирішенні задачі пакування важливо використовувати велику варіативність алгоритмів для досягнення оптимального рішення при вирішенні конкретної ситуації.

4.3 Алгоритм додавання пода у кластер з поточним навантаженням

У підрозділі 4.1 розглянуто алгоритми створення кластера, коли в ньому немає поточного навантаження, тобто, вирішено задачу планування ресурсної ємності. Проте, виникають ситуації, коли необхідно внести зміни до вже існуючого кластера, наприклад, додати новий под.

Тоді задача планування описується за допомогою формул 3.5, 3.6. Тобто, поточні вузли, які мають навантаження слід розглядати як ті, що мають ємніть таку, що не дорівнює максимальній.

Вирішується задача планування ресурсної ємності при умові, що вузли в межах одного пулу мають різну ресурсну ємність. Ресурсну ємність вузла, в якому вже є навантаження слід рахувати як різницю його максимальної ємності та його поточного навантаження.

Для вирішення такої задачі пропонується використання додаткових модифікацій щодо існуючих алгоритмів, та, введення нового алгоритму. Модифікації алгоритмів: із орієнтацією на под (Pod Centric) та орієнтацією на вузол (Node Centric).

Наприклад, алгоритм Best Fit Decreasing із орієнтацією на под має насутпні кроки:

- відсортувати поди у напрямку спадання їх розміру;
- відсортувати усі вузли у порядку зростання кількості їх вільних ресурсів;

– помістити найбільший под у найменший можливий вузол;
– якщо под неможливо помістити, то відкрити новий вузол та помістити под до нього. Новий вузол додати до списку вузлів;
– на кожній ітерації – перераховувати залишок на кожному вузлі та відповідно сортувати їх.

Модифікація Best Fit Decreasing із орієнтацією на вузол має наступні кроки:

– обчислити залишкові розміри вузлів;
– обрати найменший вузол;
– для обраного вузла: поміщати найбільші поди зі списку, поки вони вміщаються у даний вузол;
– якщо залишилися поди, то до додати новий вузол, повторити етапи алгоритму.

У роботі [20] пропонується алгоритм із балансування вузлів. Основною ідеєю такого алгоритму є те, що як тільки елемент було додано до вузла, алгоритм буде намагатися помістити наступні поди у наступні вузли, щоб запобігти переповненню вузлів дуже рано.

Діагарму активності для алгоритму планування ресурсної ємності при умові наявності деякого навантаження у кластері показано у додатку Д.

Виділяється три основні типи активностей:

– підготовка алгоритму: ініціліазація даних – зчитування усіх под, які необхідно додати до кластера, валідація усіх значень. Зчитування цільового розміру вузла для формування кластеру. Також, на даному етапі – ініціалізувати початкові значення розмірів вузлів. Початкові значення розмірів вузлів формується як різниця його максимальної ємності та його поточного навантаження. На даному етапі формується масив із вузлів різного розміру;
– планування ресурсної ємності – виконання запропонованих алгоритмів, та знаходження найкращого алгоритму. Для кожного алгоритму – виконати його модифікацію із орієнтацією на под, та з орієнтацією на вузол. Для простости, завжди береться порядок сортування за спаданням;

– аналіз результатів. Аналізуються результати, та, якщо результати надали розподілення подів між вузлами, то, надалі, поди розташовуються на вузлах за заданим розподілом.

4.4 Алгоритм вибору пулу для пода

Як було показано у розділах 2 та 3, Kubernetes має підтримку декількох пулів вузлів. У кожному пулі знаходяться вузли із однаковою ресурсною ємністю та однаковою конфігурацією. Нові вузли, що додаються до даного пулу також мають ту саму ресурсну ємність та ту саму конфігурацію, як і попередні.

Важливою частиною для планування ресурсної ємності є правильне розподілення под між вузлами.

Для вирішення задачі розподілення под між вузлами пропонується алгоритм вибору вузла, виходячи із розміру пода, та зовнішніх конфігурацій вибору. Вибор має проходити за чіткими критеріями: якщо под підпадає під задані критерії розташування на вузлі, то, надалі, виконується його розташування за алгоритмами, наведеними у підрозділах 4.1 та 4.3.

Таке розділення под є природнім, адже для бізнесу важливо логічно відділяти поди як за їх цільовим призначенням, так і за їх розміром, щоб не виникали ситуації перенасичення подами великого вузла малими подами, чи, навпаки, коли один невеликий вузол тримає лише один под.

Тоді, виходячи з вищеописаного, алгоритм розподілення под за трьома пулами вузлів має наступні етапи:

– якщо под задовольняє критеріям розташування у пулі із найменших вузлів – розташувати под у першому пулі;
– якщо под задовольняє критеріям розташування у пулі із найбільших вузлів – розташувати под у середньому пулі;
– якщо под задовольняє критеріям розташування у пулі із найбільших вузлів – розташувати под у останньому пулі;

– якщо под не задовольняє критеріям жодного пулу, то вивести повідомлення про помилку та не розташовувати под у кластері.

Критерії розміщення формуються як:

– мінімальне та максимальне значення за запитами по CPU;
– мінімальне та максимальне значення за запитами по RAM;

Критерії розподілення між пулами не можуть перетинатися, задля досягнення детермінованості алгоритму вибору пулу для розташування поду.

4.5 Висновки до розділу

У даному було розроблено алгоритми планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації. Алгоритми планування базується на таких евристиках, як Next Fit, First Fit, Best Fit, Worst Fit та варіаціях даних алгоритмів із рідним порядком обробки под. Також, запропоновано гібридний алгоритм.

Результати експериментів показують те, що алгоритми показують результати, близькі до теоретичних мінімумів (за кількістю використаних вузлів), а лідерами серед алгоритмів є алгоритми із групи гібридних алгоритмів. Також, алгоритми показують високу утилізацію ресурсів, що говорить про можливість їх широкого використання. Сортування под у більшості випадків покращує результати виконання алгоритмів.

Побудовано діаграми активностей для алгоритмів онлайн та офлайн планування, що покращує сприйняття даних алгоритмів.

5 ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

Для валідації результатів використання запропонованих у розділі 4 алгоритмів, створено спеціальний веб-застосунок, який інкапсулює у себе програмну реалізацію усіх евристичних алгоритмів, надає моделі даних основних сутностей, таких, як вузол Kubernetes, под, тощо, реалізує бізнес логіку складних процесів планування.

У даному розділі наведено обґрунтування та вибір технологій для реалізації веб-застосунку, показано загальну структуру проекту, показано основні етапи розробки.

Також, для кращого сприйняття застосунку, у даному розділі спроектовано діаграму компонентів застосунку, діаграму класів, діаграму послідовності виклику тих чи інших функцій застосунку із відповідним позначенням викликів зовнішніх інтерфейсів Kubernetes, діаграму прецедентів та діаграму розгортання.

5.1 Вибір технологій реалізації

Для розробки програмного забезпечення планування ресурсної ємності кластера Kubernetes для заданого навантаження в умовах контейнерної віртуалізацій була обрана мова програмування Python [21] та фреймворк Flask [22] для створення веб-застосунку.

Мову програмування Python обрано через те, що вона є дуже зручною для створення прототипів та реалізації різноманітних алгоритмів, моделей, а також є велика кількість готових бібліотек, фреймворків для вирішення широкого спектру задач, швидких обчислень, роботи із Kubernetes, роботи із багатьма хмарними технологіями тощо. Також, Python має широкі можливості для опису даних та моделей.

Мова Python має динамічну типізацію, зручний синтаксис, велику кількість вбудованих структур, є простою для створення власних структур – все це робить

мову програмування Python зручною для швидкого створення прототипів, реалізації різних математичних моделей.

Для мови Python було обрано середу розробки PyCharm. PyCharm розроблено компанією JetBrains на базі IntelliJ IDEA. PyCharm є інтегрованою середою розробки, та працює під більшістю операційних систем, а саме: Windows, Linux, MacOS.

Для підвищення портативності застосунку, пришвидшення процесу його розробки використовується технологія Docker.

5.1 Структура проекту

Загальну структуру проекту показано на рисунку 5.1. Основними складовими проекту є:

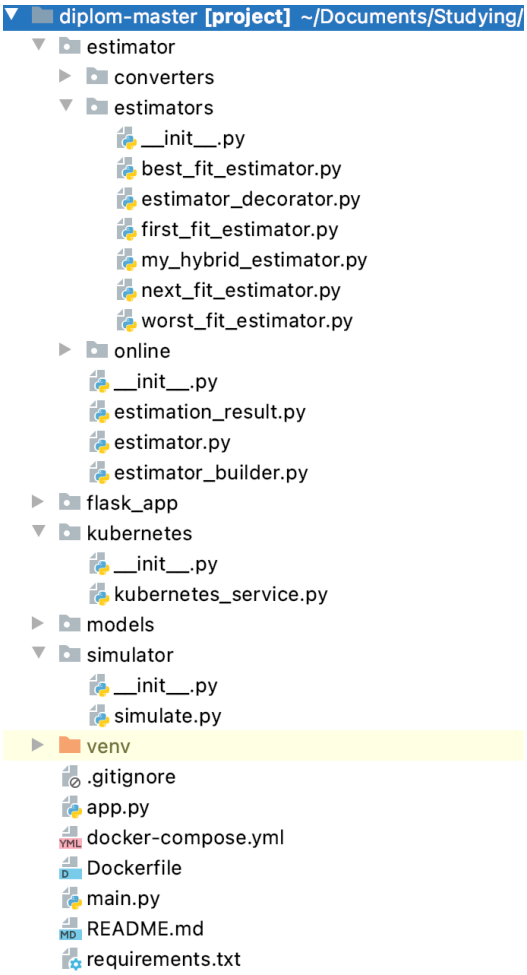


Рисунок 5.1 – Загальна структура проекту

- пакет estimator, який інкапсулює усю логіку планування ресурсної ємності для заданого навантаження. Включає підпакети для офлайн та онлайн планування;
- пакет flask_app інкаплсулює усю бізнес логіку клієнтського застосунку, реазізує інтерфейси керування, описує методи використання застосунку;
- пакет kubernetes інкапсулює логіку роботи із Kubernetes – комунікацю. із кластером, авторизацію запитів, витягування необхідної інформації тощо;
- пакет models містить усі описи моделей застосунку, такі як модель опису вузла Kubernetes, поду, ресурсної ємності, моделей запитів та відповідей тощо;
- пакет simulator містить генератор експериментів для валідації алгоритмів. Цей пакет використовувався у розділі 4 для проведення експериментів.
- App.py – головний файл застосунку. Реалізує логіку старту самого застосунку;
- Dockerfile – файл опису зображення для застосунку;
- docker-compose.yml – додатковий файл роботи із Docker, який полегшує процес розробки;
- requirements.txt – файл, який містить усі залежності проекту.

Запуск проекту виконується через docker-compose, або ж через ручну побудову зображення проекту із використанням Dockerfile, або ж автоматизований запуск через docker-compose.yml.

Зміст Dockerfile надано нижче.

```
FROM python:3.8

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .
```

```
CMD ["gunicorn", "app:app", "--log-level=INFO",
    "--threads=1", "--workers=1", "-b", "0.0.0.0:8000"]
```

Декларується, що як базове зображення використовується Python, версії 3.8. Надалі, вказується, що робочою директорією є /app. За допомогою команди pip3 install -r requirements.txt встановлюються усі необхідні залежності для проекту, а за допомогою команди COPY . . у зображення копіюються усі вихідні файли проекту.

Проект запускається командою CMD, яка викликає запуск процесу gunicorn із вказівкою який саме застосунок запускати.

5.2 Діаграма розгортання системи

Діаграма розгортання призначена для представлення загальної конфігурації або топологіїї розподіленої програмної системи і містить зображення розміщення різних артефактів на окремих вузлах системи [23].

Діаграма розгортання призначена для візуалізації компонентів і елементів системи, які існують на стадії її безпосереднього виконання. При чому представляються лише ті компоненти-екземпляри програми файлів або динамічних бібліотек, які можуть виконуватися. Інші компоненти не показані на схемі розгортання.

Діаграма розгортання містить графічні зображення процесорів, пристрої, процеси та зв’язки між ними. На відміну від логічних діаграм, діаграма розгортання є єдиною всієї розроблюваної системи, оскільки вона має повністю відображати усі особливості.

При розробці схем розгортання переслідуються наступні цілі:

- вказати фізичні вузли системи, на яких потрібно розмістити виконувані компоненти програмної системи;
- показати фізичні зв’язки між компонентами системи;
- виявити вузькі місця в системі та переконфігурувати її топологію для досягнення необхідної продуктивності

Діаграму розгортання розроблюваної системи показано у додатку Е. Детальний опис діаграми розгортання наведено нижче.

Діаграму логічно розділено на дві частини: застосунок та Kubernetes. Для застосунку використовуються два вузли: фізична машина клієнта та, безпосередньо сервер застосунку.

Фізична машина користувача включає в себе такі компоненти:

– веб-браузер користувача. Через веб браузер користувач може надсилати запити до системи, переглядати відповіді від сервера у форматі зручному для читання;
– API клієнт. Якщо клієнт хоче використовувати розроблений застосунок у автоматизованому вигляді, використовуючи будь-яку мову програмування, то він використовує API клієнт.

Комунікація із сервером застосунку проходить шляхом надсилання HTTP/HTTPS запитів. Тобто, основний метод передачі команд від клієнта – інтернет. Такий підхід забезпечує високу надійність та можливість легко масштабувати застосунок.

Наразі, сервер застосунку надає публічний API, проте, є потенціальна можливість використовувати додаткову бібліотеку для комунікації із API. Перевагою є те, що можливо розробляти та оновлювати бібліотеку незалежно від сервера, що дозволяє розділити обов’язки під час розробки та підтримки усієї системи вцілому.

Сервер застосунку приймає HTTP/HTTPS запити на одни із публічних портів серверу. Надалі, пакети за допомогою реверс проксі потрапляють безпосередньо до застосунку.

Спочатку, запит потрапляє до Gunicorn [24] сервісу. Gunicorn – це HTTP-сервер Python WSGI для UNIX. Сервер Gunicorn широко сумісний з різними веб-фреймворками, просто реалізований, не потребує великих ресурсів сервера і досить швидкий.

Gunicorn має наступні переваги:

– підтримує WSGI, Django та Flask;

– автоматичне управління робочим процесом;
– проста конфігурація Python;
– кілька конфігурацій воркеру;
– різні серверні хуки для розширюваності;
– сумісний з Python 3.x > = 3.5.

Надалі, запит оброблюється безпосередньо Flask сервером. В залежності від вказаного у запиті шляху, обирається відповідний обробник даного запиту. Надалі, якщо необхідно, обробник викликає бібліотеку роботи із Kubernetes.

Бібліотека роботи із Kubernetes використовує HTTP/HTTPS запити для виконання запитів безпосередньо до головного вузла Kubernetes.

Вузол сервера застосунку показано на рисунку 5.2.

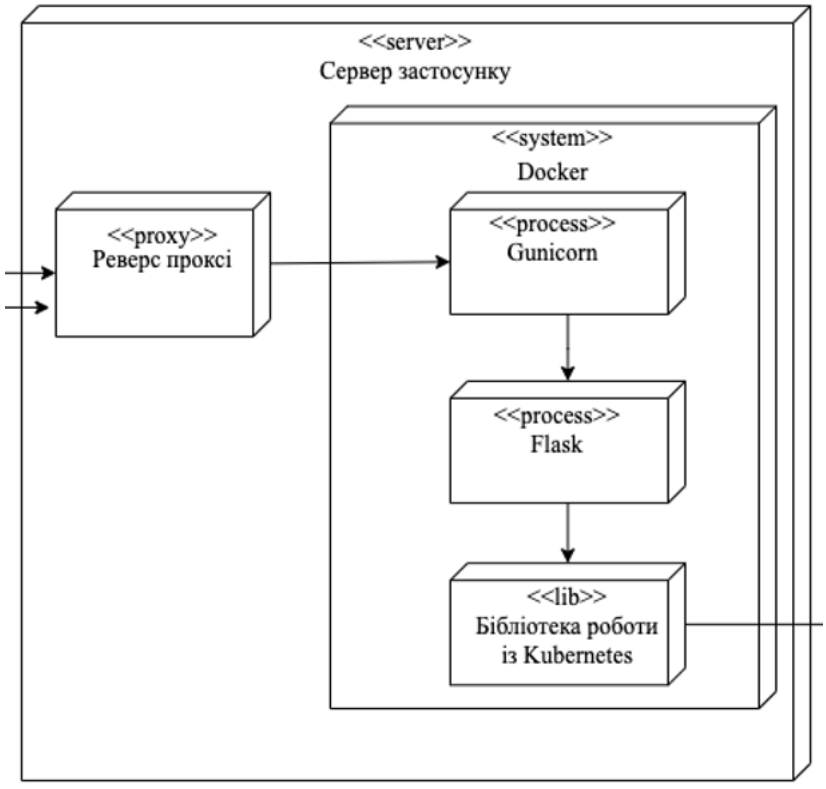


Рисунок 5.2 – Вузол сервера застосунку

Надалі, у головному вузлі Kubernetes виконуються наступні процедури:

– робиться відповідний запит до бази даних etcd. Якщо це операція читання, то виконується запит на читання. Якщо ж це операція запису, то виконується вставка, або оновлення даних у базі;
– робиться запит до планувальника, якщо була операція додавання пода, або ж додавання нового вузла;
– робиться запит до Controller Manager для моніторингу нових сутностей, якщо була операція додавання.

Для комунікації із робочими вузлами використовуються HTTP/HTTPS запити. На робочих вузлах розгорнуто kube-proxy через який запити надходять на веб сервер робочого вузла. Один головний вузол може комунікувати із необмеженою кількістю робочих вузлів.

Вузол Kubernetes Master зображено на рисунку 5.3.

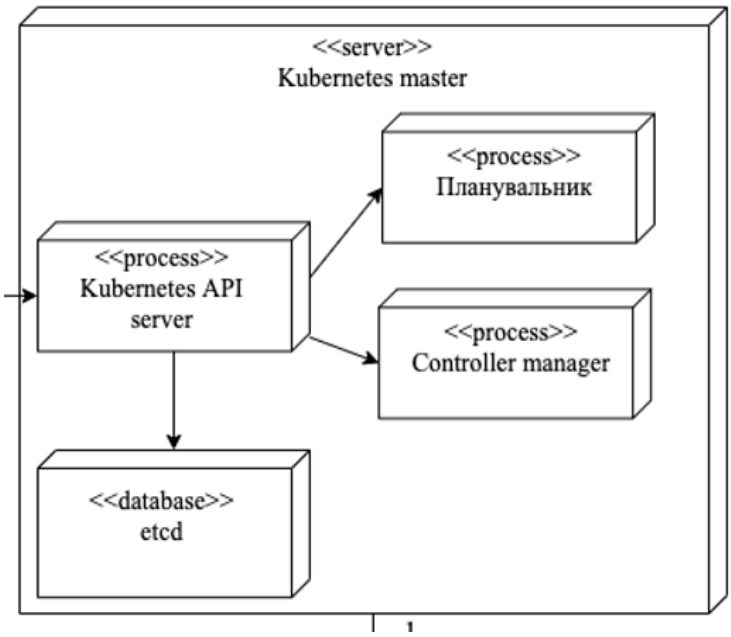


Рисунок 5.3 – Вузол Kybernetes Master

Отже, використано три основних вузла: вузол клієнта, вузол веб сервера та вузол Kubernetes master.

5.3 Діаграма прецедентів системи

Діаграма прецедентів – це графічне зображення можливої взаємодії користувача з системою. Діаграма прецедентів показує різні варіанти використання та різні типи користувачів, які система має, і часто супроводжуватимуться також іншими типами діаграм. Варіанти використання представлені колами або еліпсами. Акторів часто зображують у вигляді фігурок.

Основними елементами діаграми прецедентів є:

- актор – це одна, або декілька ролейей, які використовуються при взаємодії із прецендентами або сутностями (система, підсистема або клас). Актором може бути людина або інша система, підсистема або клас, які представляють інші сутності;
- прецедент – позначає виконувані системою дії (може включати можливі варіанти), що призводить до спостерігаємих акторами результатів. Надпис може бути ім'ям або описом того, що робить система, а не як. Під час виконання сценарію актори обмінюються із систематичними повідомленнями.

У діаграму прецедентів включено два основних відношення:

- включення – специфікує той факт, що деякий варіант використання містить поведінку, яку включено до іншого сценарію використання;
- розширення – визначає взаємозв'язок одного варіанту використання з деяким іншим варіантом використання, функціональність або поведінку якого задіюється першим не завжди, а тільки при виконанні деяких додаткових умов.

Відношення включення та відношення розширення позначають так, як показано на рисунку 5.4.

Діаграму прецедентів для розроблюваної системи показано у додатку Ж. У даній діаграмі використання визначено п’ять основних варіантів використання, які відповідають відповідній кінцевій точці Flask застосунку.

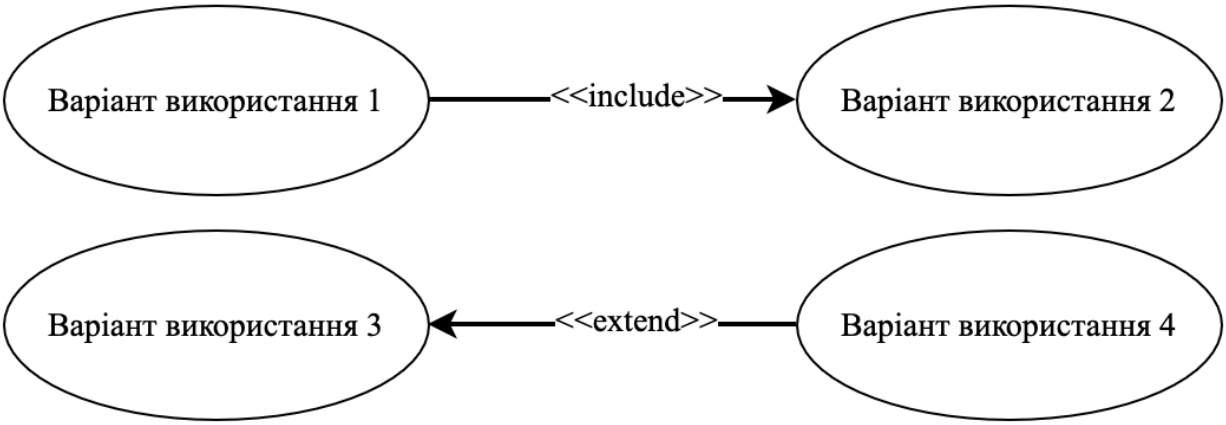


Рисунок 5.4 – Приклад відношень включення та розширення

Перший варіант використання – отримати інформацію по кластеру. Використовується тоді, коли користувач хоче отримати усю наявну інформацію про кластер.

Даний варіант використання включає в себе такі пункти:

– отримання загальної інформації, такої як версія Kubernetes, стан кластеру, кількість вузлів, версії встановленого на них програмного забезпечення тощо;
– отримання інофрмації про вузли кластеру – перелік усіх вузлів кластеру, їх розміру, статусу; Також надається інформація про поди, тобто, список под на конкретному вузлі кластера;
– отримання інформації про наявні пули кластера – перелік усіх пулів вузлів кластера.

Другий варіант використання – додати конфігурацію розподілення под між пулами вузлів. Даний варіант використання включає в себе збереження наданої інформації. При виникненні помилки валідації за правилами, наведеними у розділі 4, показується повідомлення про помилку.

Третій варіант використання – отримати план ресурсної ємності для заданого навантаження. Включає в себе :

– валідацію наданих даних. Валідуються конфігурації под, які мають бути розтошовані у кластері за правилами, наведеними у пункту 4. При наявності невірної конфігурації виводиться повідомлення про помилку;
– вибір оптимального алгоритму. Для наданих под обирається найкращий алгоритм їх розташування. Найкращим є той алгоритм, що використовує найменше вузлів для розташування усіх под; Вибір прободиться за алгоритмом, наведеним у пункут 4;
– формування результатів. Формуються результати вибору оптимального алгоритму. У результати включено інформацію про кількість доданих вузлів та відношення між подом та вузлом, на якому под буде розташовано (для кожного наданого пода). При неможливості спланувати ресурсну ємність видається повідомення про помилку.

Третій варіант використання показано на рисунку 5.5.

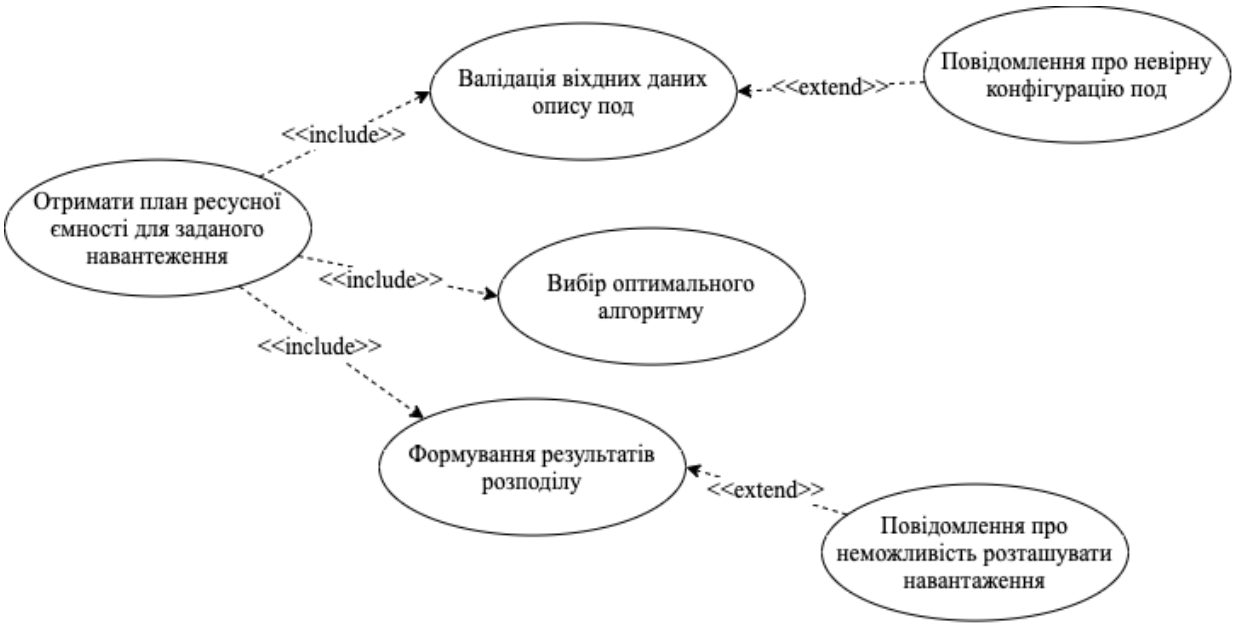


Рисунок 5.5 – Варіант використання отримання плану ресурсної ємності

Четвертий варіант використання – примінити план ресурсної ємності. План ресурсної ємності було побудовано у третьому варіанті використання. Результатом примінення плану є розташовані поди у кластері Kubernetes.

П’ятий варіант використання – додати под у існуючий кластер. Включає в себе примінення алгоритмів онлайн планування, описаних у розділі 4 та розташування вказаного пода у кластері Kubernetes.

5.4 Діаграма компонентів системи

Діаграма компонентів, показує особливості фізичного представлення всієї системи. За допомогою діаграми компонентів визначаються архітектуру розроблюваної системи, встановлюють залежності між програмними компонентами, які можуть бути представлені як вихідний, бінарний і виконуваний код.

Зазвичай, у більшості систем, компонент відповідає файлу. Основними графічними блоками даної діаграми є компоненти, інтерфейси, надані або необхідні, залежності між компонентами.

Діаграма компонентів має на меті наступні цілі:

- візуалізація загальної структури вихідного коду програмної системи;
- специфікація виконуваної програмної системи;
- забезпечення можливості багаторазового використання деяких фрагментів системи та програмного коду.

Основними елементами діаграми компонентів є:

- компонент – елемент моделі, що представляє деяку модульну частину системи з інкапсульованим вмістом, специфікація якого є взаємозамінною в його оточенні;
- наданий інтерфейс – інтерфейс, який компонент надає для свого оточення;
- необхідний інтерфейс – інтерфейс, який необхідний компоненту від свого оточення для виконання заявленої функціональності, контракту або поведінки.

Діаграму компонентів системи наведено у додатку З. Детальний опис даної діаграми наведено нижче.

Основною стартовою точкою виконання обробки запитів є компонет Gunicorn, який виступає у ролі веб сервера для Python, та вимагає від компонентів, які з ним взаємодіють реалізацію інтерфейсу Python WSGI – Python Web Server Gateway Interface.

Компонент Flask Web App реалізує інтерфейс WSGI, та є основним компонентом виконання програми. Саме з нього починається програмна обробка запитів.

Компонент Flask Web App використовує наступні пакети:

– пакет моделей запитів і відповідей користувачу. Містить основні можелі у вказаній області, реалізує їх серіалізацію та десеріалізацію;
– клієнт бізнес логіку – допоміжний пакет для виконання всієї бізнес логіки.

Завдяки такому розділенню компонентів, частина, яка відповідає за взаємодію із користувачем може розвиватися окремо від частини, яка відповідає безпосередньо за бізнес логіку.

Пакет бізнес логіки вміщує в себе логіку виклику відповідних алгоритмів планування ресурсної ємності, аналіз результатів, вибір найкращого алгоритму, роботу із Kubernetes. У своїй роботі використовує пакет роботи із Kubernetes через відповідний інтерфейс, та пакет планування ресурсної ємності через відповідний інтерфейс.

Пакет роботи із Kubernetes інкапсулює усю логіку роботи із Kubernetes – зчитування інформації з кластера, авторизацію запитів до Kubernetes, додавання нових вузлів, додавання под тощо. У своїй роботі використовує пакет роботи із Kubernetes у конкретного провайдера.

Пакет планування ресурсної ємності для заданого навантаження інкапсулює наступні елементи:

– усі класи алгоритмів офлайн планування. Містить різні варіації та можливість змінювати порядок обробки усіх под;
– усі класи алгоритмів онлайн планування.

Пакет планування ресурсної ємності використовує пакет конвертації векторів через відповідний інтерфейс.

Пакет конвертації веректорів містить у собі алгоритми конвертації багатовимірних значень у скалярні. Реалізує формули 3.7 та 3.8 для виконання розрахунків. Надає відповідний публічний інтерфейс

5.5 Діаграма класів системи

Діаграма класів – діаграма, призначена для представлення моделі статичної структури програмної системи в термінології класів об'єктно орієнтованого програмування.

Діаграма класів може відображати взаємозв'язки між сутностями предметної області: об’єкти, класи, а також описує їх внутрішню структуру із описом полів і структур, а також типу відносин: наслідування, композиція тощо. На даній діаграмі не вказується інформація про тимчасові аспекти функціонування системи.. На цьому етапі принципово знання ООП підходу і патернів проектування.

Варіанти графічного зображення класів на діаграмі класів показано на рисунку 5.2.

Відповідно до рисунку 5.6 – є три варіанти графічного зображення класів: за ім’ям класу, за ім’ям класу, його атрибутами та операціями, тільки за операціями класу, тільки за атрибутами класу.

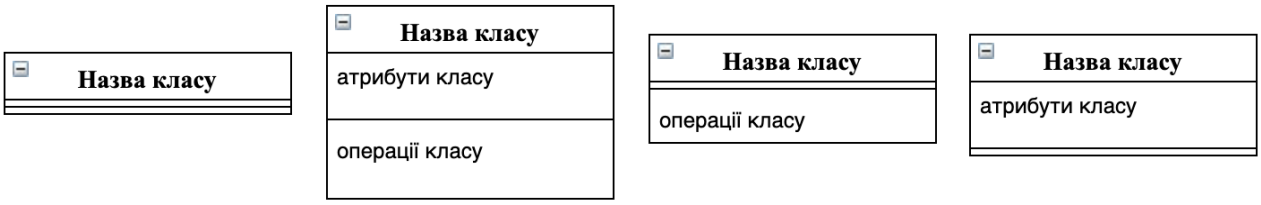


Рисунок 5.6 – Варіанти графічного зображення класів

Атрибути та операції класу мають область видимості:

- «+» загальнодоступні;
- «-» закриті
- «#» захищені

– «~» пакетні

Варіанти залежностей між класами на діаграмі класів показано на рисунку 5.7.

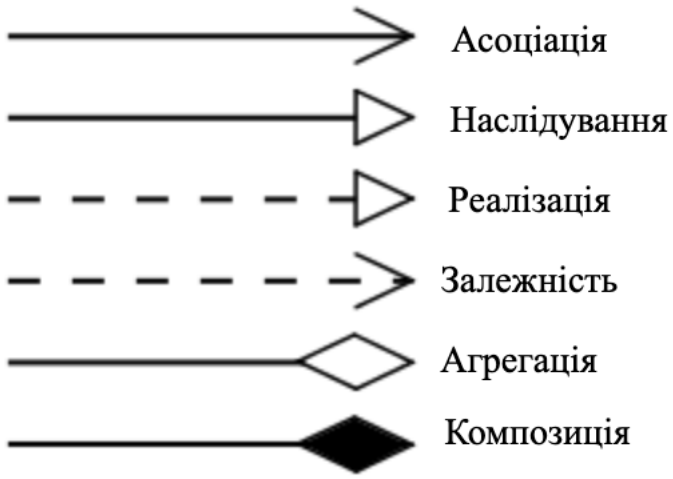


Рисунок 5.7 – Варіанти залежностей між класами на діаграмі класів

Використовується шість основних залежностей між класами: асоціація, наслідування, реалізація, залежність, агрегація, композиція. Опис кожного типу залежностей:

Асоціація. Відображає стосунки між двома класами. Використовується коли два класи повинні спілкуватися, і один (або обидва) класи містять посилання на другий. Асоціаційні відносини є сильнішими, ніж відносини залежності, вони передбачають тісніший зв'язок між сутностями:

– наслідування. Позначає що один клас наслідує інший;
– реалізація. Показує, що клас реалізує деякий інтерфейс;
– залежність. Зв'язок залежності передбачає, що два елементи залежать один від одного. Використовується для відображення того, що один клас взаємодіє з іншим, отримує екземпляр цього класу як параметр методу. Порівняно з асоціацією, залежність відносин слабша;
– агрегація. Передбачає пов’язаність двох класів, це призводить до більш детальної інформації щодо характеру стосунків: дитина може існувати незалежно від батьків;

– композиція. Передбачає пов’язаність двох класів і додає наступні деталі: всередині композиції суб’єкти сильно залежать від цілого. Об'єкти типів створюються разом і мають спільний життєвий цикл.

Діаграму класів для розроблюваної системи наведено у додатку И.

Основним класом, який реалізує більшу кількість бізнес логіки шляхом виклику інших класів є клас Client. Клас Client відповідає на основні запити користувача так інкапсулює всередині себе основну логіку.

Клас Client має лише одне поле kubernetes_service, для роботи із Kubernetes кластером.

Методи класу Client:

– get_cluster_indo() – звертається до Kubernetes через kubernetes_service та видає усю інформацію про кластер;
– get_estimation() – запускає алгоритм планування ресурсної ємності, обирає найкращий алгоритм серед тих, що були запущені. Формує результат для кінцевого коритсувача;
– apply_estimation() – розташовує поди на вузлах після планування. Результатом виконання даного методу є розташовані поди у кластері;
– add_pools_configuration() – локально зберігає налаштування щодо розташування под на вузлах;
– add_new_pod() – додає новий под до кластеру. Використовує алгоритми онлайн планування ресурсної ємності.

Основними класами даних у застосунку є Node, Pod та Capacity.

Клас Capacity інкапсулює ресурсну ємність у вигляді процесорного часу та кількості оперативної пам’яті. Має 2 властивості:

– cpu – ресурсна ємність процесорного часу;
– ram – ресурсна ємність оперативної пам’яті;

Клас Pod представляє собою под у Kubernetes, та має наступні властивості:

– name – унікальну ім’я пода;
– requests – ресурси, які є необхідними для даного пода;

Клас Node представляє собою вузол у Kubernetes, та має наступні властивості:

– name – унікальне ім’я вузла;
– labels – список міток вузла;
– capacity – ресурсна ємність вузла;
– pods – усі поди, які розташовано на вузлі;
– remaining_capacity – та ресурсна ємність, яка є вільною для використання.

Методи класу Node:

– generate_with_capacity(capacity) – створює новий екземпляр вузла із випадковим ім’ям та заданою ресурсною ємністю. Такий вузол не буде прив’язано до реально об’єкта у Kubernetes – лише абстракція;
– add_pod(pod) – додає под до власного списку под;
– remove_pod(pod) – видаляє под до власного списку под;
– can_fit(pod) – перевіряє, чи можна розмістити наданий под на поточному вузлі

Взаємозв’язок між класами даних показано на рисунку 5.8.

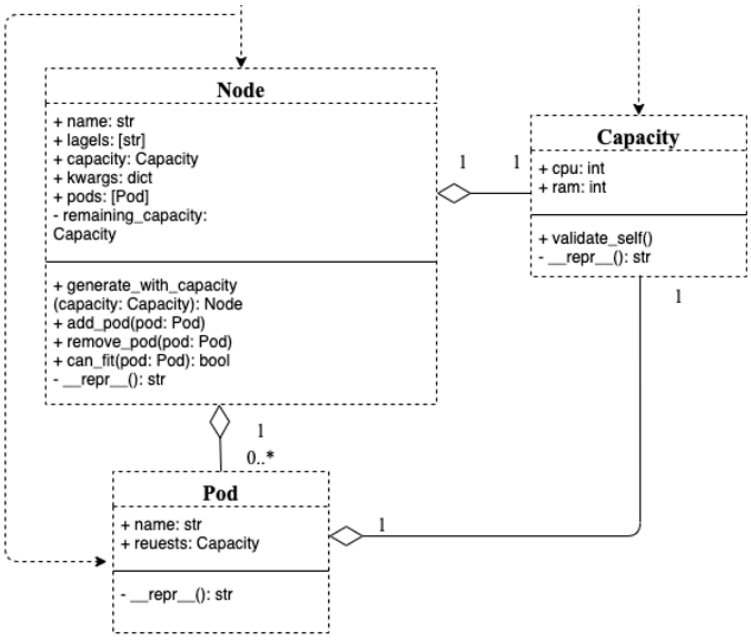


Рисунок 5.8 – Взаємозв’язок між класами даних

Node та вузол мають по одному екземпляру Capacity. У випадку для Node, capacity описує його ресурсну ємність, тобто, скільки максимально він має

ресурсів. У випадку із Pod – показує скільки ресурсів необхідно для розташування поду на вузлі кластера.

Node має відношення агрегації із подами, що показує природу їх зв’язку – поди розташовуються на вузлі, та кожен вузол може мати нульову, або додатню кількість под.

Базовим класом планувальника є клас Estimator. Він має єдиний метод estimate, який обчислює необхідну кількість вузлів, щоб розташувати задане навантаження, та надає розташування под на вузлах. Для планування приймає наступні аргументи:

- node_capacity – цільовий розмір вузла. При плануванні буде розраховуватися, що усі вузли мають такий розмір;
- pods – поди, які необхідно розташувати;
- vector_converter – конвертер, який перетворює Capacity у скалярне значення. Використовується алгоритмами, що потребують порівняння елементів Capacity.

Estimator є базовим класом, тому усі алгоритми, наведені у пункті 4 реалізовано у сабкласах Estimator, а саме: FirstFitEstimator, BestFitEstimator, WorstFitEstimator, HybridEstimator.

Для зміни порядку обробки под у вищенаведених класах, створено окремий декоратор, що викликає базовий алгоритм із зміненим порядком под. Декоратор містить один екземпляр класу із базовим алгоритмом та потребує, щоб було вказано порядок сортування под.

Відношення між класами планувальників показано на рисунку 5.8.

Класи типу VectorConverter надають алгоритми із перетворення об’єктів типу Capacity у скалярне значення. Від класу VectorConverter унаслідувано класи MulVectorConverter та SumVectorConverter.

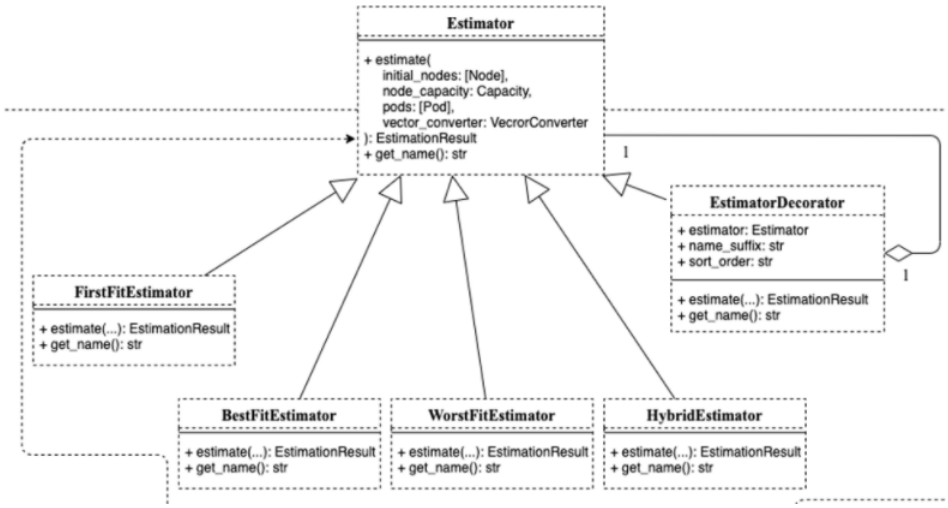


Рисунок 5.8 – Відношення між класами планувальників

VectorConverter надає єдиний метод: convert, який приймає capacity, та коефіцієнти вектору. SumVectorConverter переводить Capacity до скалярного вигляду шляхом складання ресурсів, помножених на відповідні коефіцієнти, а MulVectorConverter переводить Capacity до скалярного вигляду шляхом перемноження ресурсів, помножених на відповідні коефіцієнти. За замовчуванням коефіцієнти рівні одиниці

5.6 Діаграма послідовності обробки запитів

Діаграма послідовності – діаграма, призначена для відображення порядку взаємодії між елементами моделі програмної системи в термінології ліній життя і повідомлень між ними.

Графічно діаграма послідовності має два основних виміри. Перший – зліва направо у вигляді вертикальних ліній, кожна з яких відповідає лінії життя окремого учасника взаємодії. Другий – вертикальна вісь, спрямована зверху вниз. Перший варіант використання – отримати інформацію по кластеру. Використовується тоді, коли користувач хоче отримати усю наявну інформацію про кластер.

Реалізація взаємодії моделюється за допомогою повідомлень, які передаються між різними лініями життя. Повідомлення зображуються у вигляді стрілок різної форми і утворюють деякий порядок щодо часу своєї передачі. Ті повідомлення, що

розташовані на діаграмі послідовності вище, передаються раніше ніж ті, які розташовані нижче.

Основними елементами діаграми послідовності є:

– лінія життя представляє одного індивідуального учасника взаємодії або окрему взаємодіє сутність. Прямокутники на лінія життя позначаються діяльність об’єкта, або виконання їм деякої функції;
– повідомлення – елемент моделі, призначений для подання окремої комунікації між лініями життя деякого взаємодії.

Основні позначення для діаграми послідовності наведено на рисунку 5.9.

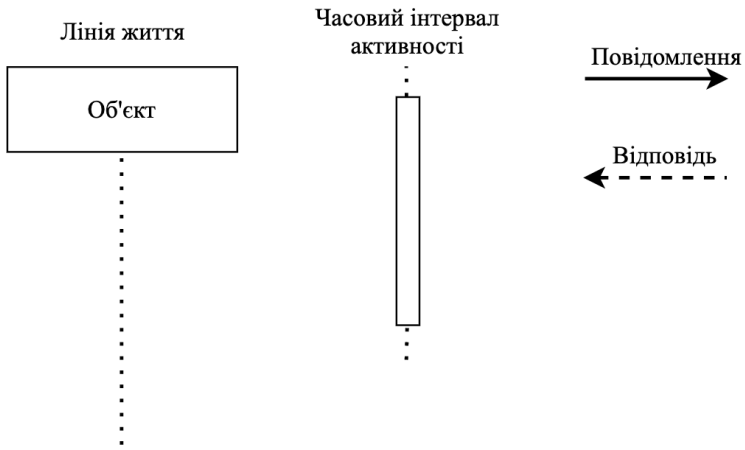


Рисунок 5.9 – Основні позначення для діаграми послідовності

Діаграму послідовності обробки запитів від користувача наведено у додатку К. На даній діаграмі послідовності показано дві основні послідовності обробки запитів: обробка запиту на планування та обробка запиту на примінення планування.

Обробка запиту на планування включає в себе використання трьох сутностей: користувач, Flask Server та модуль планування ресурсної ємності, та має наступні етапи:

– користувач надсилає свою команду у застосунок, де, надалі, команда оброблюється за допомогою Flask сервера;
– Flask сервер валідує запит користувача. Перевіряє, чи дотримано усіх необхідних критерії при формуванні запиту;

- надалі, виконується бізнес логіка та виконуються основні алгоритми планування – у модулі планування ресурсної ємності;
- результат виконання алгоритмів передається назад;
- серед результатів виконання алгоритмів знаходиться найкращий;
- формується результат, та надсилається користувачеві.

На даному етапі користувач може ознайомитися із планом ресурсів, можливо, може внести зміни до корисного навантаження.

Обробка запиту на примінення плану включає в себе використання трьох сутностей: користувач, Flask Server, модуль роботи із Kubernetes, Kubernetes API Server, Kubernetes Scheduler, та має наступні етапи:

- користувач надсилає свою команду у застосунок, де, надалі, команда оброблюється за допомогою Flask сервера;
- запит користувача аналізується, та формується розподіл;
- надалі, формуються команди до модуля роботи із Kuberentes, який відповідає за комунікацію із кластером Kubernetes;
- модуль роботи із Kubernetes формує запити на додавання под до Kubernetes API Server та надсилає їх;
- kubernetes API Server оброблює запит, та подає команди на свій внутрішній планувальник;
- результат від Kubernetes планувальника передається назад до kubernetes API Server;
- kubernetes API Server повертає результат виклику API до модуля роботи із Kubernetes;
- модуль роботи із Kuberentes повертає до Flask Server результати викликів API Kubernetes;
- Flask Server формує результат для користувача на базі отриманих відповідей від API сервера та надсилає сформовану відповідь користувачу.

Обробка запитів є лінійною, передбачає послідовний обмін повідомленнями між компонентами всієї системи.

5.7 Модуль симуляції запитів

Для перевірки правильності роботи алгоритмів планування було створено модуль для симуляції запитів.

Даний модуль відповідає за дві основні функції: створення тестового навантаження та виклик усіх варіацій алгоритмів для перевірки їх правильності та готовності бути використаними.

Створення тестового навантаження проходиться за критеріями:

– граничні значення CPU для под;
– граничні значення RAM для под;
– кількість под.

Лог генерації тестового навантаження показано на рисунку 5.10.

```
Згенеровано под: Pod `Pod_bc2e0c65-6583-4564-9365-7c67ca6213f9`: Requests: Capacity: cpu[175], ram[431]
Згенеровано под: Pod `Pod_b6b3873e-20fe-42df-b2fa-4ce010d3efef`: Requests: Capacity: cpu[130], ram[271]
Згенеровано под: Pod `Pod_efc0d4ae-86e4-4d0f-94c3-ad350734b2a9`: Requests: Capacity: cpu[187], ram[658]
Згенеровано под: Pod `Pod_cc397cf0-d6c1-4864-887d-222f28b0e631`: Requests: Capacity: cpu[213], ram[456]
Згенеровано под: Pod `Pod_59f22089-c788-4cdb-b750-541ce2bf2bf0`: Requests: Capacity: cpu[233], ram[426]
Згенеровано под: Pod `Pod_d9c94c81-39b1-42e9-8dcd-cc3b325b822c`: Requests: Capacity: cpu[232], ram[401]
Згенеровано под: Pod `Pod_3954f850-1193-4982-b672-2e513e661367`: Requests: Capacity: cpu[104], ram[556]
Згенеровано под: Pod `Pod_358ae649-90df-4965-8a86-dfaaef3eb15e`: Requests: Capacity: cpu[340], ram[307]
Згенеровано под: Pod `Pod_805b1133-2341-4393-88c8-fd52c2af43be`: Requests: Capacity: cpu[251], ram[670]
Згенеровано под: Pod `Pod_7be77a06-c7d0-423d-8d9e-6c5da85aa9af`: Requests: Capacity: cpu[190], ram[414]
Згенеровано под: Pod `Pod_c3f1f7df-1697-40d7-9392-2de3e8f668f5`: Requests: Capacity: cpu[237], ram[397]
Згенеровано под: Pod `Pod_57490416-8828-4e64-b7a5-1006d0dd7690`: Requests: Capacity: cpu[339], ram[267]
```

Рисунок 5.10 – Лог генерації тестового навантаження

Як видно з рисунку 5.10 створюються поди з випадковими іменами, та з ресурсними запитами у межах встановлених границь. У даному випадку – CPU: [100, 350] та RAM [256, 712]. Створюється така кількість под, яка була задану у конфігурації.

Надалі, виконуються алгоритми розподілення под для цільового розміру вузла. Для кожного алгоритму виконуються три варіації: із природнім порядком под, із реверсивним порядком под та прямим порядком под.

Використовуються наступні алгоритми:

– NextFitEstimator();

– FirstFitEstimator();
– BestFitEstimator();
– WorstFitEstimator();
– MyHybridWorstEstimator();
– MyHybridBestEstimator().

Тобто, виконується 18 алгоритмів – шість базових із трьома варіаціями порядку вказування под.

Лог тесту показано на рисунку 5.11.

```
Estimator: Next Fit Decreaing. Used: 863. CPU utilization: 77.98 %. RAM utilization: 82.69 %
Estimator: Next Fit Increasing. Used: 865. CPU utilization: 77.8 %. RAM utilization: 82.5 %
Estimator: Next Fit As Is. Used: 869. CPU utilization: 77.44 %. RAM utilization: 82.12 %
Estimator: First Fit Decreaing. Used: 841. CPU utilization: 88.73 %. RAM utilization: 93.32 %
Estimator: First Fit Increasing. Used: 855. CPU utilization: 83.21 %. RAM utilization: 87.55 %
Estimator: First Fit As Is. Used: 851. CPU utilization: 87.39 %. RAM utilization: 92.26 %
Estimator: Best Fit Decreaing. Used: 766. CPU utilization: 87.86 %. RAM utilization: 93.16 %
Estimator: Best Fit Increasing. Used: 815. CPU utilization: 82.58 %. RAM utilization: 87.56 %
Estimator: Best Fit As Is. Used: 774. CPU utilization: 86.95 %. RAM utilization: 92.2 %
Estimator: Worst Fit Decreaing. Used: 765. CPU utilization: 87.97 %. RAM utilization: 93.28 %
Estimator: Worst Fit Increasing. Used: 815. CPU utilization: 82.58 %. RAM utilization: 87.56 %
Estimator: Worst Fit As Is. Used: 783. CPU utilization: 85.95 %. RAM utilization: 91.14 %
Estimator: Hybrid Worst Fit Decreaing. Used: 767. CPU utilization: 87.74 %. RAM utilization: 93.04 %
Estimator: Hybrid Worst Fit Increasing. Used: 767. CPU utilization: 87.74 %. RAM utilization: 93.04 %
Estimator: Hybrid Worst Fit As Is. Used: 767. CPU utilization: 87.74 %. RAM utilization: 93.04 %
Estimator: Hybrid Best Fit Decreaing. Used: 765. CPU utilization: 87.97 %. RAM utilization: 93.28 %
Estimator: Hybrid Best Fit Increasing. Used: 815. CPU utilization: 82.58 %. RAM utilization: 87.56 %
Estimator: Hybrid Best Fit As Is. Used: 773. CPU utilization: 87.06 %. RAM utilization: 92.32 %
Lower Bound: 714
```

Рисунок 5.11 – Лог тесту алгоритмів

5.8 Висновки до розділу

У даному розділі показано основні етапи проектування та розробки програмного застосунку для планування ресурсної ємності. Мовою програмування для реалізації застосунку обрано Python, а цільовим фреймворком – Flask із реалізацією веб-сервера на Gunicorn.

Розроблено діаграму розгортання системи, діаграму прецедентів, діаграму компонентів, діаграму класів застосунку та діаграму послідовності обробки запитів від користувача.

ВИСНОВКИ

Дисертаційна робота присвячена вирішенню актуальної науково-практичної проблем планування ресурсної ємності для заданого навантаження в умовах контейнерної віртуалізації.

В роботі проведено аналіз особливостей контейнерної віртуалізації. За результатами аналізу можна дійти висновку про широкий потенціал використання даної технології для вирішення широкого спектру задач.

Для роботи із контейнерами найчастіше використовуються контейнерні окестратори, які виконують функцію автоматизованої координації, конфігурації, та управління їх сукупністю. Проте, такі системи не мають механізму планування ресурсної ємності.

Розроблено математичну модель проблеми. Проблему класифіковано як проблему багатовимірного пакування та багатовимірного пакування із контейнерами різного розміру. Тобто, необхідно оптимально розмістити надані елементи – віртуальні контейнери на фізичних, або віртуальних машинах.

Побудовано алгоритм планування, із використанням існуючих евристичних алгоритмів для вирішення проблеми багатовимірного пакування, а також запропоновано нову, власну гібридну евристику. Розроблено власні методи переходу між багатовимірним вектором ресурсів у скалярне значення.

Результати експериментальних досліджень показують наближеність використовуваних алгоритмів до оптимуму, а також високу утилізацію ресурсів.

Розроблено програмний застосунок мовою програмування Python, із використання фреймворків Flask та Gunicorn для надання кінцевому користувачеві доступу до алгоритмів. Наведено основні діаграми, які описують застосунок та систему.

ПЕРЕЛІК ПОСИЛАНЬ

1. Опис технології віртуалізації [Електронний ресурс] – Режим доступу до ресурсу: https://azure.microsoft.com/en-us/overview/what-is-virtualization/.

2. History of Virtualization [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/cd/E26996_01.

3. Oracle VM Hypervisor [Електронний ресурс] – Режим доступу до ресурсу: https://www.oracle.com/virtualization/.

4. Прайсинг AWS EC2 [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/ec2/pricing/.

5. Офіційний портал Docker [Електронний ресурс] – Режим доступу до ресурсу: https://www.docker.com/.

6. Алгоритм Raft [Електронний ресурс] – Режим доступу до ресурсу: https://neerc.ifmo.ru/wiki/index.php?title=Raft.

7. Amazon ECS – Офіційний сайт [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/ecs/.

8. Amazon Fargate – Офіційний сайт [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/fargate/.

9. Офційний сайт Nomad [Електронний ресурс] – Режим доступу до ресурсу: https://www.nomadproject.io/.

10. Сайт опису Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

11. Аналіз порівняння популярності контейнерних окрестраторів [Електронний ресурс] – Режим доступу до ресурсу: https://phoenixnap.com/blog/container-orchestration-tools.

12. Цитовцева А. С. Аналіз доступності мікросервісів на базі системи управління та оркестрації контейнерів Kubernetes / А. С. Цитовцева, О. О. Сопов. // Проблеми інформатизації та управління. – 2021. – №1. – С. 91–96.

13. Сопов О. О. Особливості масштабування контейнерного навантаження на базі системи Kubernetes / О. О. Сопов, А. С. Цитовцева. // Технічні науки та технології. – 2021. – №23. – С. 103–108.

14. Теленик С. Ф. Управление распределением виртуальных машин в ЦОД / С. Ф. Теленик, А. І. Ролік, Е. В. Жаріков.

15. Жаріков Е. В. Динамічне розміщення віртуальних машин на основі навчання з підкріпленням в хмарних центрах обробки даних / Е. В. Жаріков, А. А. Коваль, Р. А. Терентьєв. // Наукові вісті Далівського університету. - 2017. - № 13

16. Fatima A. Virtual Machine Placement via Bin Packing in Cloud Data Centers / A. Fatima, W. Hussain, M. Bilal. // MDPI Electronics. – 2018. – №7.

17. Coffman E. An introduction to bin packing [Електронний ресурс] / E. Coffman, J. Csirik, D. Johnson. – 2004.

18. Leeuwen J. Handbook of Theoretical Computer Science. Vol. A, Algorithms and complexity / Jan van ed Leeuwen. – Amsterdam: Elsevier, 1998.

19. Механізм розподілу пода на вузол [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/.

20. Gabay M. Variable size vector bin packing heuristics - Application to the machine reassignment problem / M. Gabay, S. Zaourar. // HAL. – 2013.

21. Офіційний сайт Python [Електронний ресурс] – Режим доступу до ресурсу: https://www.python.org/.

22. Офіційний сайт Flask [Електронний ресурс] – Режим доступу до ресурсу: https://flask.palletsprojects.com/en/1.1.x/.

23. Буч Г. Язык UML. Руководство пользователя / Г. Буч, Д. Рамбо, И. Якобсон. – М.: ДМК Пресс, 2006. – 496 с.

24. Офіційний сайт Gunicorn [Електронний ресурс] – Режим доступу до ресурсу: https://gunicorn.org/.

Додаток А

Перелік публікацій

UDC 001.1

The 8th International scientific and practical conference "Science and education: problems, prospects and innovations" (April 28-30, 2021) CPN Publishing Group, Kyoto, Japan. 2021. 866 p.

ISBN 978-4-9783419-5-2

The recommended citation for this publication is:
Ivanov I. Analysis of the phaunistic composition of Ukraine // Science and education: problems, prospects and innovations. Proceedings of the 8th International scientific and practical conference. CPN Publishing Group. Kyoto, Japan. 2021. Pp. 21-27. URL: https://sci-conf.com.ua/viii-mezhdunarodnaya-nauchno-prakticheskaya-konferentsiya-science-and-education-problems-prospects-and-innovations-28-30-aprelya-2021-goda-kioto-yaponiya-arhiv/.

Editor
Komarytskyy M.L.
Ph.D. in Economics, Associate Professor

Collection of scientific articles published is the scientific and practical publication, which contains scientific articles of students, graduate students, Candidates and Doctors of Sciences, research workers and practitioners from Europe, Ukraine, Russia and from neighbouring coutries and beyond. The articles contain the study, reflecting the processes and changes in the structure of modern science. The collection of scientific articles is for students, postgraduate students, doctoral candidates, teachers, researchers, practitioners and people interested in the trends of modern science development.

e-mail: kyoto@sci-conf.com.ua

homepage: ***https://sci-conf.com.ua***

УДК 004.94

ПОКРАЩЕННЯ АЛГОРИТМУ РОЗПОДІЛУ НАВАНТАЖЕННЯ МІЖ ВІРТУАЛЬНИМИ МАШИНАМИ У КЛАСТЕРІ KUBERNETES

Сопов Олексій Олександрович
студент-магістрант
Національний технічний університет України
«Київський політехнічний інститут
імені Ігоря Сікорського»
м. Київ, Україна

Анотація: Використання технологій контейнерної віртуалізації та мікросервісної архітектури стає все більш популярним в останній час. Система Kubernetes забезпечує відносно легкі для адаптації механізми керування контейнеризованими застосунками, керування ресурсами кластера. Kubernetes у своїй базові конфігурації використовує достатньо прості та примітивні алгоритми розподілення контейнерного навантаження між машинами у кластері, що призводить до низької утилізації ресурсів та необхідності розширювати кластер.

У даній роботі розглянуті основні принципи роботи Kubernetes, зокрема алгоритм роботи планувальника. Запропоновано більш оптимальні алгоритми для розподілення контейнерного навантаження між машинами у кластері.

Експерименти показують, що використання алгоритмів, запропонованих у даній статті, покращує утилізацію ресурсів та зменшує кількість необхідних машин у кластері.

Ключові слова: Kubernetes, контейнерана віртуалізація, алгоритм, утилізація ресурсів, кластер ресурсів, планування ресурсів.

У складному обчислювальному контексті хмарні обчислення можуть значно покращити рівень використання ресурсів. В хмарній архітектурі

використання контейнерної віртуалізації, такої як Docker, набуває дедалі більшого поширення [1]. Задача керування контейнерами при невеликій їх кількості є доволі тривіальною, та полягає в розгортанні контейнерів на віртуальних машинах із заданням лімітів за процесорним часом та кількості оперативної пам'яті. Проте, зазвичай, неможливо досягти масштабованості при використанні такого підходу. Для вирішення таких проблем було розроблено багато систем, таких як Docker Swarm, Amazom ECS, Kubernetes, MicroKubernetes тощо. Найбільш популярною на даний час є система Kubernetes, адже вона надає найбільш широкі можливості, та є публічним проектом, що дозволяє його змінювати під свої потреби.

Однією з найбільших переваг використання Kubernetes є можливість розгортати контейнерне навантаження на декількох віртуальних машинах одночасно, що дозволяє досягти масштабованості та підвищити надійність розроблюваних програмних продуктів. Контейнерне навантаження у Kubernetes описується за допомогою сутності, під назвою под (англ. Pod), та усі ресурси об'єднуються у кластер ресурсів із віртуальних машин, які у Kubernetes названо як нода (англ. Node) [2].

План розміщення ресурсів у Kubernetes відіграє важливу роль в оптимізації продуктивності, що визначає оптимальне розміщення / зіставлення под з нодами із різними цілями та обмеженнями. В загальному сенсі, проблема розміщення ресурсів пов'язана з класичним пакуванням контейнерів, де мета - це упакувати або розмістити або відобразити набір предметів / об'єктів різного розміру в мінімальну кількість контейнерів з наперед відомою місткістю.

Оскільки проблема пакування контейнерів є NP-повною проблемою, точні алгоритми потребують більше часу, щоб отримати рішення, отже, цю проблему можна ефективно апроксимувати, використовуючи алгоритми наближення, такі як жадібні алгоритми, генетичні алгоритми для досягнення результатів. Вони забезпечують рішення, яке, хоча і дуже добре в більшості випадків, може бути не оптимальним рішенням, але майже оптимальним.

У Kubernetes є планувальник за замовчуванням для пошуку вузла для

нещодавно створеного пода. Планувальник використовує пріоритетну чергу, в якій головний под буде запланований спочатку до вибраного вузла. Стратегія планування за замовчуванням включає три етапи: предикативній, пріоритетний та етап вибору.

На кроці предикат планувальник перевіряє всі доступні вузли та фільтрує вузли які відповідають вимогам под відповідно до деяких предикатів, таких як CheckNodeConditionPred та PodFitsResources; тоді планувальник за замовчуванням буде обчислювати оцінку кожного доступного вузла на другому кроці; Пріоритет розраховується за формулою (1). Надалі, нода із найбільшим пріоритетом використовується для розміщення пода [3].

$$score = \sum \big(type\big((capacity - requested) * 10\big) \,/\, capacity\big) / \, 3$$
$$type \in \{gpu, cpu, mem\}\} \qquad (1)$$

Тобто, Kubernetes вирішує задачу із пакування контейнерів, яка має велику кількість рішень, окрім запропонованої стандартної.

Нехай, S = S1, S2, ... набір нод, які мають однаковий розмір V. Нехай, a1, a2, ... і набір под, які потрібно упакувати в под. Завдання полягає у виявленні цілочисельної кількості нод B, так щоб $\sum_{i \in S_k} a_i \leq V, \forall k = 1, 2..B$. Оптимальне рішення є в знаходженні найменшого B. Формування задачі для лінійного програмування наведено у формулі 2.

$$\min B = \sum_{i=1}^{n} y_i \qquad (2)$$

де B ≥ 1,

$\sum_{j=1}^{n} a_j x_{ij} \leq V_{y_i}, \forall i \in (1,2,..n)$,

$\sum_{i=1}^{n} x_{ij} = 1, \forall j \in (1,2,..n)$,

$y_i \in (0,1), \forall i \in (1,2,..n)$, $x_{ij} \in (0,1), \forall i \in (1,2,..n), \forall j \in (1,2,..n)$

Для вирішення даної проблеми пропонується використання евристичних алгоритмів для вирішення задачі пакування контейнерів, таких як Next Fit, First Fit, Best fit, First Fit Decreasing, Best Fit Decreasing [4, 5] та Dot Product [6].

Поди буде розподілено за допомогою вищенаведених алгоритмів до нод у

системі Kubernetes. Результати буде порівняно із оригіальним алгоритмом розподілення у Kubernetes.

Вхідними даними для експерименту є випадкові величини розміру под із розплоділів [100, 400]. [1, 1000], [200, 800], [50, 200] та [25, 100] із випадковою кількістю. Метриками для аналізу є: кількість под, використаних для розподілення усіх под та відсоток утилізації ресурсів.

Алгоритмами для порівняння є: стандартний алгоритм Kubernetes (K8S), Next Fit (NF), First Fit (FF), Best fit (BF), First Fit Decreasing (FFD), Best Fit Decreasing (BFD) та Dot Product (DP). Результати по кількості використаних под наведено на рисунку 1.

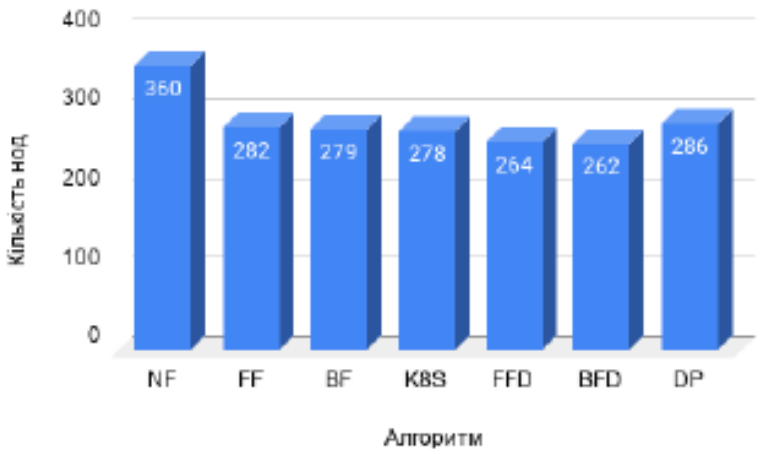


Рис.1 – Порівняння кількості використаних нод для пакування усіх под

Як видно з порівняльного графіку кількості використаних под для пакування усіх заданих под, то, в середньому, алгоритму показують схожі результати. Проте, порівнюючи запропоновані алгоритми із стандартним алгоритмом пакування у Kubernetes, видно що алгоритми FFD, BFD дають кращі показники, що може бути основоположним для зміни базового алгоритму пакування у Kubernetes. Для них кількість використаних нод є на 14 та 16 менше, та означає значну економію ресурсів кластера Kubernetes.

Порівняння відсотку утилізації ресурсів при використанні різних алгоритмів наведено на рисунку 2.

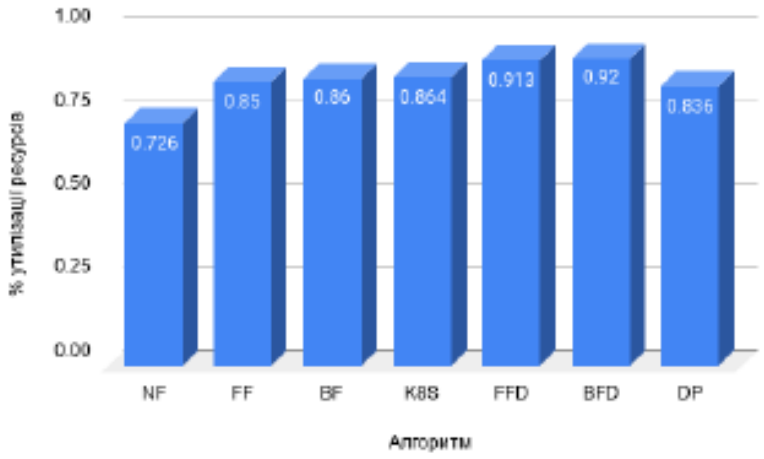


Рис.2 – Порівняння відсотку утилізації ресурсів на нодах

Результати є аналогічними до тих, що було отримано при порівнянні кількості використаних: алгоритми FFD та BFD показують найкращі результати. Аналізуючи результати із кількості використаних нод та порівняння відсотку утилізації на нодах, можна дійти висновку, що стандартний алгоритм пакування у Kubernetes показує хороші результати, проте запропоновані алгоритми, зокрема FFD та BFD показують кращі результати.

Висновки: Було розглянуто основні принципи роботи Kubernetes, зокрема алгоритм роботи планувальника. Досліджено стандартний алгоритм пакування ресурсів у Kubernetes, та запропоновано ряд алгоритмів для оптимізації розподілення под серед нод Kubernetes. Алгоритми First Fit Decreasing та Best Fit Decreasing показують кращі результати, ніж стандартний алгоритм пакування у Kubernetes, що є підставою для використання даних алгоритмів як базових.

СПИСОК ЛІТЕРАТУРИ

1. Sam Newman. Building Microservices: Designing Fine-Grained System / Sam Newman – O'Reilly, 2015. – 251 с.

2. Офіційний портал Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/.

3. Планувальник Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler.

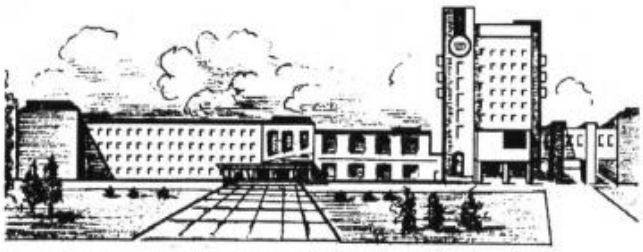
4. Xia B. Discrete Applied Mathematics / B. Xia, Z. Tan. // Tighter bounds of the First Fit algorithm for the bin-packing problem. – 2010.

5. Zehmakan N. Bin Packing Problem: Two Approximation Algorithms / N. Zehmakan. // CoRR. – 2015.

6. Anand A. Virtual machine placement optimization supporting performanceslas, / A. Anand, J. Lakshmi, S. Nandy. // Cloud Computing Technology and Science. – 2013.

ISSN 2411-5363 (print)
ISSN 2519-4569 (online)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЧЕРНІГІВСЬКА ПОЛІТЕХНІКА»



ТЕХНІЧНІ НАУКИ ТА ТЕХНОЛОГІЇ

НАУКОВИЙ ЖУРНАЛ

№ 1(23)

Чернігів 2021

УДК 62:67.05
Т38
DOI: 10.25140/2411-5363-2021-1(23)

Друкується за рішенням вченої ради Національного університету «Чернігівська політехніка» (протокол № 3 від 29.03.2021 р.). Науковий журнал «Технічні науки та технології» внесено до переліку наукових фахових видань України, затвердженого Наказом Міністерства освіти і науки України від 11.07.2019 р. № 975, відповідно до якого журналу надано категорію «Б».

Технічні науки та технології : науковий журнал / Національний університет «Чернігівська політехніка». – Т38 Чернігів : НУ «Чернігівська політехніка», 2021. – № 1(23). – 266 с.

У цьому випуску журналу «Технічні науки та технології» вміщено статті, присвячені теоретичним та експериментальним дослідженням у науковому напрямі «Технічні науки» за спеціальностями: прикладна механіка, матеріалознавство, машинобудування, інформаційно-комп'ютерні технології, електроенергетика, електротехніка та електромеханіка, хімічні та харчові технології, будівництво та геодезія. Статті прорецензовані провідними вченими у відповідних галузях знань.

Журнал «Технічні науки та технології» буде корисним для науковців, науково-педагогічних працівників, докторантів, аспірантів та студентів технічних спеціальностей закладів вищої освіти.

УДК 62:67.05

Заснований у 1996 році. Свідоцтво про державну реєстрацію КВ № 24601-14541 ПР від 07.09.2020

ЗМІСТ

РОЗДІЛ І. ПРИКЛАДНА МЕХАНІКА, МАТЕРІАЛОЗНАВСТВО ТА МАШИНОБУДУВАННЯ

РОЗДІЛ ІІ. ІНФОРМАЦІЙНО-КОМП'ЮТЕРНІ ТЕХНОЛОГІЇ

РОЗДІЛ ІІІ. ХІМІЧНІ ТА ХАРЧОВІ ТЕХНОЛОГІЇ

ТЕХНІЧНІ НАУКИ ТА ТЕХНОЛОГІЇ № 1(23), 2021
TECHNICAL SCIENCES AND TECHNOLOGIES
УДК 62-503.5
DOI: 10.25140/2411-5363-2021-1(23)-103-108

Олексій Сопов, Анна Цитовцева

ОСОБЛИВОСТІ МАСШТАБУВАННЯ КОНТЕЙНЕРНОГО НАВАНТАЖЕННЯ НА БАЗІ СИСТЕМИ KUBERNETES

*На сьогодні технологія контейнеризації набуває широкого поширення, та проблема масштабування є однією з найбільш важливих для підвищення продуктивності систем. **Kubernetes** є передовим рішенням для керування контейнерами, проте проблема масштабування залишається слабо описаною та дослідженою. У даній роботі досліджено особливостей масштабування контейнерного навантаження на базі системи Kubernetes. Розкриті основні операції для досягнення горизонтального та вертикального масштабування. Описані властивості масштабованості системи Kubernetes. Стаття є оглядовою.*

***Ключові слова:** Kubernetes, мікросервіс, контейнерне навантаження, масштабування, Docker.*
Рис.: 4. Бібл.: 6.

Актуальність теми дослідження. На сьогодні виконання програмних застосунків в умовах контейнерної віртуалізації надає ряд переваг, як з фінансової сторони, так і зі сторони продуктивності, відмовостійкості та швидкості роботи. Тому, сфера розробки програмного забезпечення з використанням технології контейнеризації набуває широкого поширення в багатьох сферах бізнесу

З огляду на часті зміни навантаження на програмні продукти і різні умови їх функціонування, завдання масштабування корисного навантаження стає все більш актуальним. Проблема масштабування є однією й найбільш важливих для розширення обсягу виконаних задач за період часу, тому важливо використовувати правильні схеми масштабування застосунків.

Kubernetes є передовим рішенням при роботі із контейнерами та забезпечує відносно легкі для адаптації механізми керування, проте проблема масштабування залишається слабо описаною та дослідженою, що підвищує складність переходу бізнесу на контейнерні технології, зокрема, із використанням Kubernetes.

Постановка проблеми. В останній час триває перехід від класичної монолітної архітектури побудови систем до сервіс-орієнтованої, або ж, найчастіше, до мікросервісної архітектури задля забезпечення високого рівня доступності та відмовостійкості. Однією з найбільших переваг використання такого підходу є можливість масштабування для досягнення адекватної реакції, з огляду на часті зміни навантаження та різні умови функціонування системи в цілому.

Беручи до уваги переваги використання контейнерів, окремим мікросервісом найчастіше виступає один, або група об'єднаних контейнерів, які виконують ту, чи іншу задачу. Проте, керування такою системою з плином часу стає доволі складним, та вимагає значних зусиль. Сервіс Kubernetes [1] виконує більшість задач із керування контейнерами, їх життєвим циклом та версіями, тому є одним із найзручніших у даній області.

У Kubernetes корисне навантаження та інфраструктура концептуально розділені, тому є можливість виконувати масштабування на кожній складовій окремо, що призведе до появи більш ніж двох операцій для досягнення масштабування. Хоча система Kubernetes є достатньо гнучкою, проте вона була створена нещодавно, та такі особливості масштабування із розділенням концепцій є досі не описаними.

Аналіз останніх досліджень і публікацій. З аналізу літературних джерел можна дійти висновку, що в останній час з'явилась загальна тенденція міграції до хмари та використання віртуальних машин [2]. Проблема оптимального масштабування із використанням віртуальних машин вирішена достатньо широко [3, 4]. Архітектурний стиль мікропослуг та використання контейнерного навантаження привернули значну увагу, особливо на базі системи Kubernetes.

Виділення недосліджених частин загальної проблеми. Можна зробити висновок, що Kubernetes надає значні переваги у швидкості розробки та розгортанні застосунків на базі контейнерів, проте проблема масштабування до цього часу не отримувала значної уваги з боку наукових кіл.

Мета статті є дослідження особливостей масштабування контейнерного навантаження на базі системи Kubernetes. Розкрити основні операції для досягнення горизонтального та вертикального масштабування із розділенням концепцій корисного навантаження та інфраструктури. Описати властивість масштабованості системи Kubernetes.

Виклад основного матеріалу. У класичному розумінні масштабування застосунків поділяється на вертикальне та горизонтальне. Ці вектори масштабування є основоположними та мають різні варіації у різних системах.

Горизонтальне масштабування полягає у збільшенні кількості одиниць, що містять у собі додаток. Виконується розбиття системи на більш дрібні структурні компоненти та рознесення їх по окремим обчислювальним одиниця, або їх групам, і збільшення кількості одиниць, що паралельно виконують одну і ту ж функцію.

Вертикальне масштабування полягає у збільшенні ресурсів одиниці, на якій виконується застосунок. Тобто, збільшення продуктивності кожного компонента системи з метою підвищення загальної продуктивності.

При роботі в умовах контейнерної віртуалізації необхідно розуміти основні змінні, які можуть використовуватися для горизонтального та вертикального масштабування. У даному випадку одиницею, що виконує застосунок є, безпосередньо, контейнер, який розгорнуто із деякого зображення контейнера, тобто деяка ізольована оболонка вказаного програмного коду та його залежностей.

Горизонтальне масштабування у випадку із контейнерами полягає в наступному: додається новий вузол, а саме, додається новий контейнер, який розпочато із того самого зображення, з якого розпочато і перший контейнер. Контейнер може бути розгорнуто на будь-якій фізичній, або віртуальній машині, де досягнуті необхідні умови, наприклад, встановлено Docker daemon при роботі із системою контейнерної віртуалізації Docker [5].

Вертикальне масштабування полягає в зміні мінімальної/максимальної кількості ресурсів для контейнера. Можливість обмежити використання ресурсів окремим контейнером ж є однією з найважливіших складових контейнерної віртуалізації. Під ресурсами мається на увазі: процесорний час (CPU), кількість оперативної пам'яті (RAM), ресурси диску (iops). Обмеження ресурсів є необхідністю, адже необхідно контролювати максимальну кількість, яку використовує контейнер, щоб не допустити ситуації, коли один контейнер займає більшу частину ресурсів. За замовчуванням кількість ресурсів у контейнера необмежена, проте є можливість це налаштовувати за допомогою параметрів --cpus, --memory, --device-read-bps тощо.

Незважаючи на простоту розгортання контейнеру на будь-якій обчислювальній одиниці за допомогою спеціалізованих програмних засобі, таких як Docker, перед розробниками зазвичай стають більш складні задачі: підтримка великої кількості таких контейнерів, підтримка контейнерів різного розміру, різного навантаження та їх горизонтальне та вертикальне масштабування.

Для вирішення більшості вищенаведених задач були розроблені платформи, такі як Kubernetes, що значно полегшують роботу із великою кількістю контейнерів та їх контролем. Основні структурні компоненти Kubernetes зображено на рисунку 1 (задля простоти більшу частину, яка не приймає участь у масштабуванні було опущено).

Тобто, контейнери виконуються у середині так званих под (англ. Pod), що є найменшими одиницями керування у Kubernetes. Найчастіше, один под містить в собі лише один контейнер. Тобто, у Kubernetes контейнер знаходиться всередині окремої найменшої одиниці, що може бути розгорнута, пода. Саме Pod є об'єктом для керування при масштабуванні в Kubernetes.

У Kubernetes є можливість керувати ресурсами контейнера в середині поду. Аналогічно, як і у Docker можливо керувати основними ресурсами, а саме: процесорний час (CPU), кількість оперативної пам'яті (RAM), ресурси диску (iops) [6]. Проте, особливість роботи Kubernetes полягає в тому, що необхідно вказувати два параметри для ресурсів. Перший — ліміт, тобто, скільки максимально ресурсів буде використано контейнером. Другий — запит, тобто, скільки мінімально ресурсів необхідно для роботи контейнера та без яких ресурсів його не буде розгорнуто на кластері.

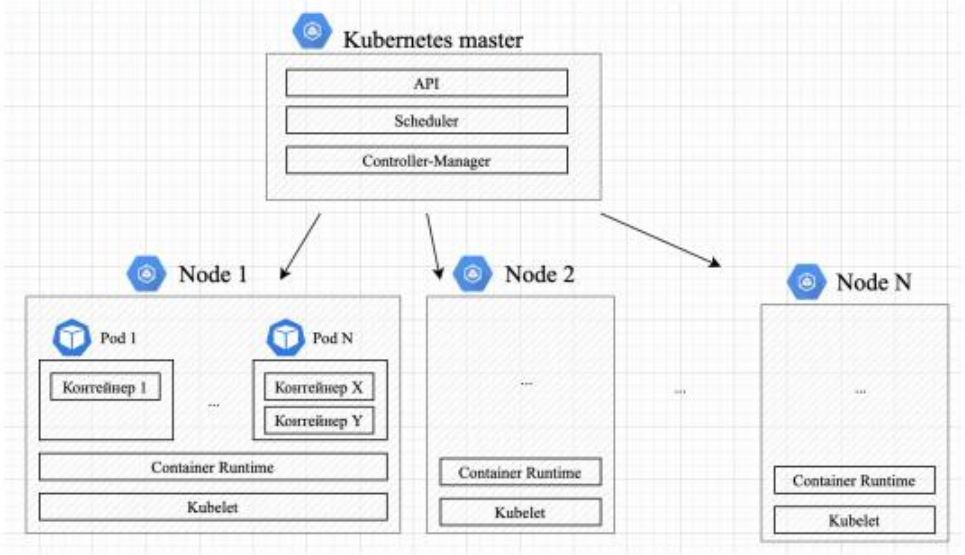


Рис. 1. Схема роботи Kubernetes

Тобто, однією з одиниць керування є ресурси контейнера всередині под: процесорний час, кількість оперативної пам'яті та ресурси диску, що можливо використовувати при виконанні масштабування.

З іншої сторони, поди розгортаються на так званих нодах. Нодами можуть виступити фізичні, або віртуальні машини, в залежності від конфігурації кластера. Фізичні та віртуальні машини мають власну ресурсну ємність, та визначають максимальну кількість под, що може бути розгорнута на них. Основними ресурсами керування для под є класичні технічні характеристики фізичної, або віртуальної машини: процесорний час, кількість оперативної пам'яті та ресурси диску.

Ресурси контейнера та ресурси ноди описують необхідні ресурси для заданого навантаження та ресурсну ємність кластера, та є основними змінними у роботі із кластером. Тому, для масштабування використано саме ці змінні.

Тобто, підсумовуючи, завдяки концепції Kubernetes у розділенні інфраструктури та додатку, два класичних типи масштабування: горизонтальне та вертикальне розширюються у двох розрізах: у розрізах інфраструктури(зміна ресурсів нод у кластері та їх кількості) та корисного навантаження(зміна ресурсів контейнерів та їх кількості).

Горизонтальне масштабування корисного навантаження в Kubernetes полягає у зміні кількості под із одним і тим самим додатком, тобто, його реплікація. Тобто, один і той самий додаток буде виконуватися паралельно, у різних подах, завдяки чому підвищується продуктивність та максимальна пропускна здатність.

Зі сторони інфраструктури горизонтальне масштабування полягає у додаванні нових нод до кластеру, на яких можливе буде виконання нових под. Тобто, збільшується кількість ресурсів, на яких можуть бути розташовані поди.

Масштабованість в цьому контексті означає можливість додавати до системи нові вузли: ноди та поди для збільшення загальної продуктивності. Це найпростіший спосіб масштабування, так як не вимагає ніяких змін в прикладних програмах, що працюють на таких системах. Схему горизонтального масштабування у Kubernetes у двох розрізах показано на рисунку 2. Як видно, основними компонентами для горизонтального масштабування є поди (у розрізі горизонтального масштабування корисного навантаження) та ноди (у розрізі горизонтального масштабування інфраструктури).

Отже, горизонтальне масштабування у Kubernetes може проводитися у двох розрізах: корисного навантаження та інфраструктурного та у двох сегментах: зміні кількості под та зміні кількості нод.

Рис. 2. Схема горизонтального масштабування у Kubernetes

Вертикальне масштабування корисного навантаження в Kubernetes полягає у зміні кількості ресурсів, які може зайняти контейнер. Тобто, у зміні кількості дозволеного використання процесорного часу, оперативної пам'яті, мережевих та дискових ресурсів.

Вертикальне масштабування у розрізі інфраструктури полягає у зміні типу віртуальної машини, на якій виконуються контейнери, тобто, зміні потужності та кількості оперативної пам'яті на даній віртуальній машині, що дозволить змінити кількість контейнерів, що можуть бути розгорнуті.

Масштабованість в цьому контексті означає можливість замінювати в існуючій системі автоматичного компоненти більш потужними і швидкими в міру зростання вимог і розвитку технологій. Цей спосіб масштабування може вимагати внесення змін до програми, щоб програми могли повною мірою користуватися дедалі більшій кількості ресурсів. Схему вертикального масштабування у Kubernetes у двох розрізах показано на рис. 3.

Отже, вертикальне масштабування у Kubernetes може проводитися у двох розрізах: корисного навантаження та інфраструктурного у двох сегментах: зміні ресурсів одиниці поду та зміні ресурсів одиниці ноди.

Дані схеми масштабування є основою для планування ресурсної ємності під задане навантаження в умовах контейнерної віртуалізації на платформі Kubernetes, тому можуть бути операціями для стратегій планування ресурсів та складовими у різних алгоритмах планування.

Рис. 3. Схема вертикального масштабування у Kubernetes

Підсумовуючи вищенаведені особливості масштабування, можна дійти висновку, що класичне масштабування (вертикальне та горизонтальне) у Kubernets виконується за допомогою чотирьох незалежних операцій (схематично такі операції показано на рис. 4):

- зміна кількості под (горизонтальне);
- зміна кількості нод (горизонтальне);
- зміна розміру контейнера, та, відповідно поду (вертикальне);
- зміна розміру машини, на якій підіймається контейнери (вертикальне).

Масштабованість Kubernetes проявляється у двох розрізах: корисного навантаження та інфраструктури. Масштабованість корисного навантаження полягає у можливості додати нові вузли, тобто, поди та ноди, а вертикального – у відповідному збільшенні ресурсів вузлів.

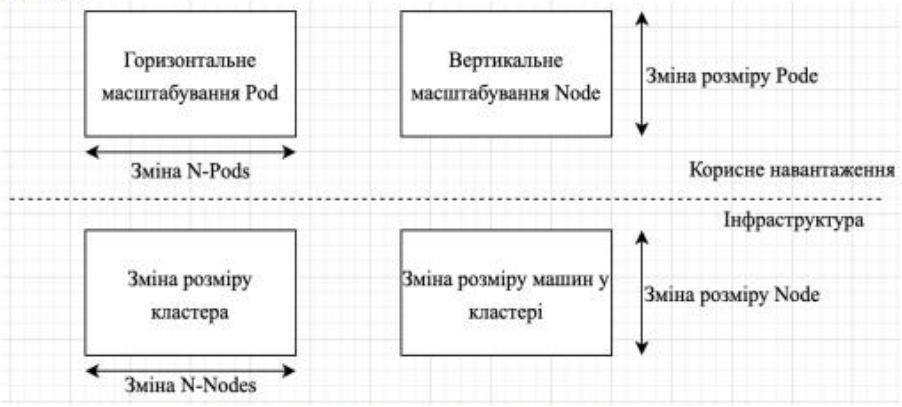


Рис. 4. Операції для досягнення масштабування у Kubernetes

Висновки. У даній статті були досліджені та описані особливості масштабування контейнерного навантаження на базі системи Kubernetes із розділенням концепцій масштабування інфраструктури та корисного навантаження.

Досліджено чотири основних операції масштабування у Kubernetes: у двох розрізах (корисного навантаження та інфраструктури) та у двох напрямках (горизонтально та вертикально). Горизонтальне масштабування досягається шляхом зміни кількості контейнерів, та, відповідно, под та зміні кількості машин у кластері, на яких можуть бути розгорнуті поди. Вертикальне масштабування полягає у зміні кількості ресурсів, як для под, так і для машин кластера.

Список використаних джерел

1. Офіційний портал Kubernetes. URL: https://kubernetes.io/.
2. Sam Newman. Building Microservices: Designing Fine-Grained System / Sam Newman – O'Reilly, 2015. – 251 с.

3. Теленик С. Ф. Управление распределением виртуальных машин в цод [Електронний ресурс] / С. Ф. Теленик, А. І. Ролик, Е. В. Жаріков. URL: https://ela.kpi.ua/bitstream/123456789/20434/1/64-13-Telenyk.pdf.

4. Жаріков Е. В. Динамічне розміщення віртуальних машин на основі навчання з підкріпленням в хмарних центрах обробки даних / Е. В. Жаріков, А. А. Коваль, Р. А. Терентьєв. // Наукові вісті Далівського університету. – 2017. – №13.

5. Офіційний портал Docker. URL: https://www.docker.com/.

6. Конфігурація ресурсів для Docker. URL: https://docs.docker.com/config/containers/resource_constraints/.

References

1. Kubernetes official portal. https://kubernetes.io/.

2. Sam Newman. Building Microservices: Designing Fine-Grained System / Sam Newman – O'Reilly, pp.251, 2015.

3. Telenyk S., Rolik A., Jarikov E., Managing the distribution of virtual machines in the data center. Retrieved from https://ela.kpi.ua/bitstream/123456789/20434/1/64-13-Telenyk.pdf.

4. Jarikov E., Koval A., Terentiev R., Dynamic deployment of virtual machines based on training with reinforcement in cloud data centers. *Scientific news of Daliv University,* №13, 2017

5. Docker official portal. https://www.docker.com/.

6. Docker resources configuration. https://docs.docker.com/config/containers/resource_constraints/.

UDC 62-503.5

Oleksii Sopov, Anna Tsytovtseva

FEATURES OF SCALING CONTAINER LOAD IN KUBERNETES SYSTEM

Today, the field of software development using container technology is becoming widespread in many areas of business, and the problem of scaling is one of the most important to achieve the expansion of the volume of tasks performed over time, so it is important to use the right scaling schemes. Kubernetes requires relatively easy-to-adapt container management mechanisms, but the problem of scaling them remains poorly described and researched, which increases the complexity of the business transition to container technologies and, in particular, the Kubernetes system.

Thanks to the separation of infrastructure and payload concepts in Kubernetes, the classic scaling methods – horizontal and vertical – are revealed in each component separately. Although the Kubernetes system is quite flexible, it has only recently been developed, and the features of scaling with concept separation have not yet been described.

Actual scientific researches and issues analysis showed that there is a general trend towards cloud migration recently. In this context, the architectural style of microservices and the use of container load has attracted considerable attention, especially based on the Kubernetes system. It can be concluded that Kubernetes has significant advantages in the speed of development and deployment of container-based applications, but the problem of scaling and features of horizontal and vertical scaling in Kubernetes has not received much attention from academia so far.

The aim of the work is to study the features of scaling the container load based on the Kubernetes system. To explain the basic operations to achieve horizontal and vertical scaling. To describe the scalability property of the Kubernetes system.

This work reveals the main architectural features of Kubernetes, the principles of working with applications in container virtualization. The main methods of regulating the resource capacity of containers are shown. The concepts of horizontal and vertical scaling in the Kubernetes system with an indication of scalability properties are revealed. The main operations for scaling in the Kubernetes system in the context of separation of payload and infrastructure concepts are given.

In this article the features of container load scaling based on the Kubernetes system with a separation of the concepts of payload and infrastructure scaling were investigated and described. This article is a review.

Keywords: *Kubernetes, microservice, application containers, scaling, docker.*

Fig.: 4. References.: 6.

Сопов Олексій Олександрович — студент-магістрант, Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського" (пр-т Перемоги, 37, м. Київ, 03056, Україна).
Sopov Oleksii — master student, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute" (37 Peremohy Avenue, 03056 Kyiv, Ukraine).
E-mail: sopov.alax.a@gmail.com
ORCID: https://orcid.org/0000-0003-0389-3070

Цитовцева Анна Сергіївна — студент-магістрант, Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського" (пр-т Перемоги, 37, м. Київ, 03056, Україна).
Tsytovtseva Anna — master student, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute" (37 Peremohy Avenue, 03056 Kyiv, Ukraine).

Сопов О., Цитовцева А. Особливості масштабування контейнерного навантаження на базі системи Kubernetes. *Технічні науки та технології.* 2021. № 1(23). С. 103-108.

УДК 62-503.5

Цитовцева А.С.,
Сопов О.О.

АНАЛІЗ ДОСТУПНОСТІ МІКРОСЕРВІСІВ НА БАЗІ СИСТЕМИ УПРАВЛІННЯ ТА ОРКЕСТРАЦІЇ КОНТЕЙНЕРІВ KUBERNETES

Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського"

Постановка проблеми в загальному вигляді

Перехід до мікросервісної архітектури вже триває. Однак, як важливий атрибут якості для послуг оператора, доступність залишається проблемою. Доступність - це нефункціональна характеристика, що визначається як обсяг відключення послуги протягом певного періоду [5]. Висока доступність досягається, коли система доступна принаймні в 99,999% випадків. Отже, загальний час простою, дозволений протягом одного року для високодоступних систем, становить близько 5 хвилин [3]. Деякі характеристики мікросервісів та контейнерів, такі як малі та легкі, природно сприяли б підвищенню доступності [1]. Kubernetes забезпечує зцілення своїх керованих мікросервісних програм. Можливість відновлення Kubernetes полягає у перезапуску невдалих контейнерів та заміні або переплануванні контейнерів, коли їхні господарі виходять з ладу. Цілюща здатність також відповідає за відновлення нездорових контейнерів, поки вони знову не будуть готові. Ці функції також би природно покращили доступність послуг, що надаються програмами, розгорнутими Kubernetes. Питання в тому, яка доступність надається цими програмами?

Аналіз останніх досліджень і публікацій

Протягом останнього десятиліття з'явилась загальна тенденція до міграції до хмари [1]. У цьому контексті архітектурний стиль мікропослуг [2] привернув значну увагу. На відміну від монолітного архітектурного стилю, мікросервісна архітектура вирішує проблеми побудови власних хмарних додатків, використовуючи переваги хмари [3]. Незважаючи на те, що цей архітектурний стиль готовий здійснити революцію в IT-галузі, до цього часу він отримував обмежену увагу з боку наукових кіл.

Мета статті

У цій роботі було оцінено програми з мікросервісною архітектурою з точки зору доступності, оскільки наша кінцева мета – забезпечити високу доступність мікросервісів. Як продовження для початкової установки в приватній хмарі та конфігурації Kubernetes за замовчуванням, було досліджено інші архітектури, конфігурації та проведено серію експериментів з Kubernetes, виміряно час відключення для різних сценаріїв відмов. Метою було дати відповідь на такі дослідницькі запитання:

- Який рівень доступності Kubernetes може підтримувати для своїх керованих мікропослуг виключно завдяки своїм властивостям відновлення?
- Який вплив додавання надмірності на доступність можна досягти за допомогою Kubernetes?
- Якої доступності може досягти Kubernetes за найефективнішої конфігурації?
- Як доступність, досягнута за допомогою Kubernetes, порівнюється з існуючими рішеннями?

Експерименти було проведено в конфігурації Kubernetes за замовчуванням, а також у найбільш реагуючій. Для кращого позиціонування та характеристики отриманих результатів було обрано порівняння з існуючим рішенням для управління доступністю, Framework Management Framework (AMF) [1], перевіреною послугою проміжного програмного забезпечення для управління високою доступністю (HA).

Виклад основного матеріалу

1. Розгортання контейнерів у кластері Kubernetes, що працює в загальнодоступній хмарі.

У цьому розділі буде розглянуто кластер Kubernetes, що складається з віртуальних машин, що працюють у загальнодоступній хмарі. Kubernetes працює на всіх віртуальних машинах і створює єдиний вигляд кластера. Одна з віртуальних машин вибрана в якості ведучої, і вона відповідає за управління вузлами. Оскільки ми стурбовані високою доступністю, нам слід розглянути кластер HA, що складається з більш, ніж одного ведучого. Однак така установка все ще є експериментальною та незрілою для Кубернетеса. Таким чином, ми вирішили піти лише з одним майстром і не допустити відмови з боку майстра. Для простоти додаток тут складається лише з одного мікросервісу. Шаблон pod для контейнерної мікросервіси, а також бажана кількість реплік включені до специфікації контролера розгортання, яка розгортається в кластері. Ми обговоримо два способи виставлення послуг у кластерах Kubernetes, що працюють у загальнодоступній хмарі.

Сервіс типу Load Balancer: архітектура для розгортання програм у кластері Kubernetes з використанням служби типу Load Balancer в загальнодоступній хмарі показана на рис. 1. На додаток до IP кластера, послуги типу Load Balancer мають зовнішню IP-адресу, яка автоматично встановлюється як IP-адреса балансира навантаження хмарного провайдера. Використовуючи цю зовнішню IP-адресу, яка є загальнодоступною, можна отримати доступ до подів ззовні кластера.

Ingress: Може бути більше однієї служби, яку потрібно піддавати зовнішньому впливу, і за принципом попереднього методу, для кожного сервіру потрібен один балансир навантаження. З іншого боку, вхідний ресурс Kubernetes може мати декілька сервісів у якості резервних копій та мінімізувати кількість балансувальних навантажень [3]. У кластері Kubernetes, що працює в загальнодоступній хмарі, контролер входу розгортається та виставляється службою типу Load Balancer [3]. Отже, запити на всі послуги, що надсилаються на балансир навантаження хмарного провайдера, отримуються контролером входу та перенаправляються на відповідну службу на основі правил, визначених у ресурсі входу

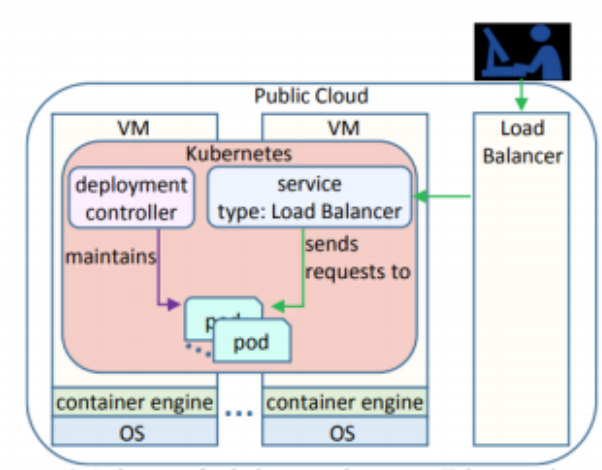


Рис. 1. Архітектура для розгортання програм на кластерах Kubernetes, що працюють у загальнодоступній хмарі

2. Розгортання контейнерів у кластері Kubernetes, що працює у приватній хмарі.

У цьому розділі буде описано налаштування експериментів, розглянуті сценарії відмов, а також показники доступності.

Ми встановили кластер Кубернетеса у приватній хмарі. Цей кластер складається з трьох віртуальних машин, що працюють у хмарі OpenStack. Ubuntu 16.04 - це ОС, що працює на всіх віртуальних машинах. Kubernetes 1.8.2 працює на всіх віртуальних машинах, а контейнерний двигун - Docker 17.09. Протокол мережевого часу (NTP) використовується для синхронізації часу між вузлами. Розгорнутою програмою є VideoLan Client (VLC). У кожному темплейті пода зазначено імедж, на якому встановлено VLC. Після розгортання пода на основі цього імеджа буде створений контейнер програми, який почне потокове передавання з файлу.

Kubernetes пропонує три рівні перевірки працездатності та механізмів для управління доступністю розгорнутих мікросервісів. По-перше, на рівні додатків Kubernetes гарантує, що програмні компоненти, що виконуються всередині контейнера, справні або за допомогою перевірки стану процесу, або заздалегідь визначених проб. В обох випадках, якщо Kubelet виявляє несправність, контейнер перезапускається. По-друге, на рівні поду Kubernetes відстежує відмови підсистеми та реагує відповідно до визначеної політики перезапуску. Нарешті, на рівні вузла, Kubernetes контролює вузли кластера через розподілені демони для виявлення відмов вузлів. Якщо вузол, що розміщує под, виходить з ладу, под переплановується на інший здоровий вузол. Що стосується цих рівнів перевірки працездатності, було зазначено три набори сценаріїв відмов. У першому наборі помилка програми спричинена помилкою процесу контейнера VLC. У другому наборі це пов'язано з відмовою процесу контейнера для подів, а в третьому наборі - через збій вузла. Для кожного набору експериментувалося з різними моделями надмірності, а також з конфігурацією Kubernetes за замовчуванням та найбільш адаптивною. Кожен сценарій повторювався 10 разів, і середнє значення вимірювань показано в Таблицях I - Таблиці V. Усі вимірювання, про які повідомляється в цьому документі, складаються у секундах.

Таблиця I – Експеримент з Кубернетисом – антинадлишкова модель та конфігурація за замовчуванням

Тригер відмови (одиниця: секунди)	Час реакції	Час репарації	Час відновлення	Час простою
Відмова контейнера VLC	0.716	0.472	1.050	1.766
Відмова контейнера пода	0.496	32.570	31.523	32.019
Відмова ноди	38.187	262.542	262.665	300.852

Час реакції: виявлено час між подією відмови, яку ми вводимо, і першою реакцією Кубернетеса, що відображає подію відмови.

Час репарації: Час між першою реакцією Кубернетеса та відновленням пода.

Час відновлення: Час між першою реакцією Kubernetes і часом, коли сервіс знову доступний.

Час простою: тривалість, протягом якої сервіс був недоступний. Він представляє суму часу реакції та часу відновлення

3. Експерименти, результати та аналіз

У цьому розділі представлено архітектури, експерименти, результати та аналіз для відповіді на питання дослідження, яке було поставлено у вступі.

Оцінка дій відновлення за допомогою конфігурації Kubernetes за замовчуванням для підтримки доступності

Рис. 2 показує архітектуру цих експериментів. Модель надмірності в даному випадку не є збитковою [2], а отже, кількість стручків у специфікації контролерів розгортання лише одна.

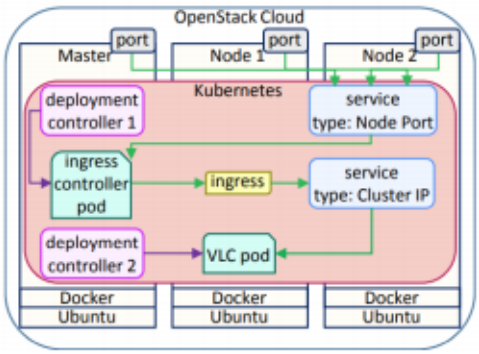


Рис. 2. Конкретна архітектура для розгортання програм за допомогою Kubernetes - модель надмірності без збитковості.

Збої в роботі через збій процесу контейнера VLC.

У цьому сценарії збій моделюється шляхом вимкнення процесу контейнера VLC з ОС. Коли контейнер VLC аварійно завершує роботу, Kubelet виявляє збій і приводить стручок до стану, коли він не отримуватиме нових запитів. У цей час, тобто час реакції, под вилучається зі списку кінцевих точок. Пізніше Kubelet перезапустить контейнер VLC, і відео почнеться з початку файлу. Цей час знаменує час ремонту. Час відновлення - це коли под знову потрапляє до списку кінцевих точок і готовий приймати запити.

Збої в роботі сервісу через збій процесу контейнера Pod.

Коли розгортається под, разом із контейнерами додатків, зазначеними в його шаблоні, створюється один додатковий контейнер, який є контейнером pod. Оскільки контейнер pod є процесом в ОС, можливо, він аварійно завершує роботу. У цьому випадку помилка моделюється шляхом вимкнення процесу контейнера підсистем з ОС. Коли процес контейнера для пода вбивається, Kubelet виявляє, що контейнер для пода відсутній, і це позначає час реакції. Коли новий под створено та запущено його контейнер VLC, відео почне транслюватися з початку файлу, і ми вважаємо под відремонтованим. Після цього Kubelet додасть новий под до списку кінцевих точок, і він буде готовий приймати нові запити, це означає час відновлення.

Збої в роботі служби через збій вузла.

У цьому випадку відмова вузла моделюється командою перезавантаження Linux на віртуальній машині, на якій розміщено под. Як вже згадувалося раніше, Kubelet відповідає за повідомлення звіту про стан вузла ведучим, і саме контролер вузла ведучого виявляє несправність вузла. Коли вузол, що розміщує под, виходить з ладу, він припиняє надсилання оновлень статусу ведучому, і майстер позначить вузол як не готовий після четвертого пропущеного оновлення стану. Цей час - час реакції. Коли вузол позначений як не готовий, підсистему VLC на вузлі планується припинити, а після його завершення буде створено новий. Час ремонту - це час запуску нового струму VLC і потокового передавання відео. Час відновлення - це коли под додається до списку кінцевих точок служби.

Результати та аналіз

Вимірювання та події цього набору експериментів наведені у таблиці I та на рис. 3 відповідно. На рис. 3 відмова контейнера VLC, контейнера pod або вузла, що розміщує Pod1, показана як перша подія. До цієї події IP-адреса Pod1 була в списку кінцевих точок, і сервіс був доступним. Після несправності сервіс стає недоступним. Однак, оскільки Kubernetes ще не виявив помилку, IP-адреса Pod1 залишається у списку кінцевих точок. Він вилучається зі списку кінцевих точок під час реакції.

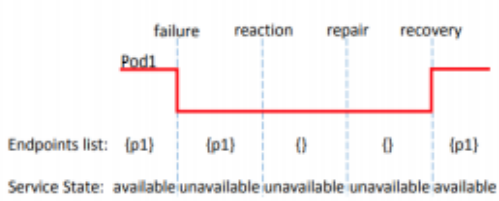


Рис. 3. Аналіз експериментів з Kubernetes з конфігурацією за замовчуванням та моделлю без збитковості – оцінка дій відновлення.

У даній архітектурі видалення IP-адреси Pod1 зі списку кінцевих точок як реакція Kubelet на збій контейнера VLC займає 0,716 секунди, а у випадку відмови контейнера pod - 0,496 секунди. Однак у разі відмови вузла, видалення IP od1 зі списку кінцевих точок, як реакція контролера вузла ведучого вимірювалася як 38,187 секунд. Причина полягає в тому, що за замовчуванням конфігурації Kubernetes, ведучому потрібно щонайменше 30 секунд, щоб виявити збій вузла. Оскільки Kubelet оновлює стан вузла кожні 10 секунд, а майстер дозволяє чотири пропущені оновлення стану, перш ніж позначити вузол як не готовий.

Час відновлення для всіх сценаріїв – це коли новий блок створюється знову і потокове відео починається знову. Як зазначається в таблиці I, час відновлення контейнера VLC або сценарії відмови контейнера пода суттєво відрізняються (0,472 секунди для першого та 32,570 секунд для останнього). Причина полягає в тому, що у випадку відмови контейнера pod, сигнал завершення надсилається до контейнера VLC, і Docker чекає 30 секунд, щоб він закінчився. Процес відновлення не розпочнеться, якщо контейнер VLC не буде припинено. Для сценарію виходу з ладу, як показано в таблиці I, час відновлення значно вищій. Причина полягає в тому, що за замовчуванням конфігурації Kubernetes, у разі відмови вузла, майстер чекає близько 260 секунд, щоб запустити новий под і відновити службу. Через такі високі терміни відновлення відключення обслуговування для сценаріїв відмови контейнера і вузла є значно вищими - 32,019 секунди та 300,52 секунди відповідно.

Висновки

В рамках даної роботи було представлено та порівняно архітектури для розгортання програм на базі мікросервісів у кластерах Kubernetes, розміщених у загальнодоступних хмарах. Проведено експерименти в хмарному середовищі, розглядаючи різні сценарії відмов, конфігурації, моделі резервування щоб оцінити Kubernetes з точки зору доступності програм, що працюють на Kubernetes. Було проаналізовано результати експериментів і виявлено, що дії для відновлення, що забезпечує Kubernetes недостатні для забезпечення доступності, особливо високої доступності. Наприклад, конфігурація Kubernetes за замовчуванням призводить до значного відключення у разі відмови вузла. Kubernetes можна переконфігурувати, щоб уникнути цього значного відключення.

Література

1. *Sam Newman.* Building Microservices: Designing Fine-Grained System. – O'Reilly, 2015. – 251 p.

2. Design Patterns: Elements of Reusable Object-Oriented Software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. – O'Reilly, 2004. – 694 p.

3. *Pethuru Raj* Chelliah. Service Discovery and API Gateways. – Essentials of Microservices Architecture, 2019

4. *Eric Evans.* Domain-Driven Design: Tackling Complexity in the Heart of Software. – Addison-Wesley, 2003. – 560 p.

5. *Robert Martin*. Clean Code: A Handbook of Agile Software Craftsmanship. – Addison-Wesley, 2008. – 465 p.

Цитовцева А. С., Сопов О. О.

АНАЛІЗ ДОСТУПНОСТІ МІКРОСЕРВІСІВ НА БАЗІ СИСТЕМИ УПРАВЛІННЯ ТА ОРКЕСТРАЦІЇ КОНТЕЙНЕРІВ KUBERNETES

У статті досліджено проблеми оркестрації та проведено експерименти, щоб оцінити доступність, яку надає Kubernetes для керованих мікропослуг. Значну увагу приділено впливу додавання надмірності на доступність програм на базі мікросервісної архітектури та проведено експерименти з конфігурацією Kubernetes за замовчуванням, а також з найефективнішою. Перехід до архітектури мікропослуг триває і зараз. При такому підході система виділяється на менші модулі, які розроблені, розроблені та масштабовані окремо для побудови віртуалізованої функції. Але для постачальників функцій доступність залишається проблемою при переході до розгортання мікросервісів. Kubernetes - це рішення, яке має базу коду з відкритим кодом, воно визначає вибір розгорнутих частин, які разом надають інструменти для побудови, підтримки, масштабування та відновлення контейнерних функцій. Таким чином, Kubernetes приховує складність організації мікросервісів, щоб забезпечити управління їх доступністю. Для початку ми оцінюємо Kubernetes, використовуючи загальну конфігурацію з точки зору доступності в хмарних налаштуваннях. Представлені архітектури для державних та приватних хмар. Ми оцінюємо доступність, яка досягається цілющою силою Кубернету. Порівняльна оцінка була проведена за допомогою Framework Management Framework (AMF), тобто запропонованого механізму, що використовується в проміжному програмному забезпеченні для управління високою доступністю. Результати дослідження демонструють, що в деяких тестах відключення служби для програм, керованих Kubernetes, є значно високим.

Ключові слова: *мікросервіси, контейнери, оркестрація, докер, доступність.*

Tsytovtseva A., Sopov O.

ANALYSIS OF THE AVAILABILITY OF MICROSERVICE BASED ON THE MANAGEMENT AND ORCHESTRATION SYSTEM OF CONTAINERS KUBERNETES

The article explores problems of orchestration and executing test of performance to assess the availability of Kubernetes for supervised services. Considerable attention is paid to the sequence of managing exorbitance, on the availability of programs with on microservice architecture and tests have been conducted with the standard settings of Kubernetes and also with the most efficient. The move to a microservices architecture continues now. In this kind of approach the system is separeted to smaller modules which are designed, developed, and scaled separetely to build virtualized function. But, for function suppliers accessibility remains an issue in the transition to deploying microservices. Kubernetes is a solution which has code base an open source, it defines a selection of deployed parts that together gives instruments for building, supporting, scaling, and recovering containerized functions. In this way, Kubernetes hides the complexity of orchestrating microservices to provide management of their approachability. To start with, we estimate Kubernetes usinggeneral configuration in terms of approachability in cloud settings. It is presented architectures for public and private clouds. We are evaluating the availability achieved by the healing power of Kubernetes. A comparative assessment was executed using the Availability Management Framework (AMF), that is a suggested mechanism used in intermediate software for high availability management. The results of the research demonstrate that in some tests, disabling the service for applications managed by Kubernetes is significantly high.

Keywords*: microservices, containers, orchestration, docker, availability.*

ДОДАТОК Б

Вихідний код засутосунку

```
from abc import ABCMeta, abstractmethod, abstractproperty
from typing import Sequence, Type

from estimator.converters.vector_converter import VectorConverter
from estimator.estimation_result import EstimationResult
from models.capacity import Capacity
from models.node import Node
from models.pod import Pod


class Estimator(metaclass=ABCMeta):
    @abstractmethod
    def estimate(
        self,
        initial_nodes: Sequence[Node],
        node_capacity: Capacity,
        pods: Sequence[Pod],
        vector_converter: VectorConverter
    ) -> EstimationResult:
        pass

    @abstractmethod
    def get_name(self):
        pass


class EstimationResult:
    def __init__(self, nodes, new_nodes_created):
        self.nodes = nodes
        self.new_nodes_created = new_nodes_created

    def __repr__(self):
        return f"Створено вузлів: {self.new_nodes_created}. Вузли: {self.nodes}"


from typing import Sequence, List, Type

from estimator.converters.vector_converter import VectorConverter
from estimator.estimation_result import EstimationResult
from estimator.estimator import Estimator
from models.capacity import Capacity
```

```
from models.node import Node
from models.pod import Pod


class BestFitEstimator(Estimator):
    def estimate(
        self,
        initial_nodes: List[Node],
        node_capacity: Capacity,
        pods: Sequence[Pod],
        vector_converter: VectorConverter
    ) -> EstimationResult:
        nodes = initial_nodes
        nodes_created = 0

        for pod in pods:
            existing_node_used = False
            best_node_index = -1
            min_score = 0

            for idx, node in enumerate(nodes):
                if not node.can_fit(pod):
                    continue

                scalar_node = vector_converter.convert(capacity=node.remaining_capacity)
                score = scalar_node

                if best_node_index == -1:
                    min_score = score
                    best_node_index = idx
                else:
                    if score < min_score:
                        best_node_index = idx
                        min_score = score

            if best_node_index != -1:
                existing_node_used = True
                nodes[best_node_index].add_pod(pod)

            if not existing_node_used:
                new_node = Node.generate_with_capacity(node_capacity)
                new_node.add_pod(pod)
                nodes.append(new_node)

                nodes_created += 1
```

```
        return EstimationResult(nodes, nodes_created)

    def get_name(self):
        return "Best Fit"


from typing import Sequence, Type

from estimator.converters.vector_converter import VectorConverter
from estimator.estimation_result import EstimationResult
from estimator.estimator import Estimator
from models.capacity import Capacity
from models.node import Node
from models.pod import Pod

SORT_DECREASING = 'Decreaing'
SORT_INCREASING = 'Increasing'
AS_IS = 'As Is'


class EstimatorDecorator(Estimator):
    def __init__(self, estimator: Estimator, name_suffix: str, sort_order: str):
        self.estimator = estimator
        self.name_suffix = name_suffix
        self.sort_order = sort_order

    def estimate(
        self,
        initial_nodes: Sequence[Node],
        node_capacity: Capacity,
        pods: Sequence[Pod],
        vector_converter: VectorConverter
    ) -> EstimationResult:
        def sort_key(px):
            scalar_px = vector_converter.convert(px.requests)
            return scalar_px

        if self.sort_order == SORT_DECREASING:
            pods = sorted(pods, key=lambda x: sort_key(x), reverse=True)
        elif self.sort_order == SORT_INCREASING:
            pods = sorted(pods, key=lambda x: sort_key(x), reverse=False)

        return self.estimator.estimate(
            initial_nodes,
```

```
            node_capacity,
            pods,
            vector_converter
        )

    def get_name(self):
        return self.estimator.get_name() + " " + self.name_suffix


from typing import Sequence, List, Type

from estimator.converters.vector_converter import VectorConverter
from estimator.estimation_result import EstimationResult
from estimator.estimator import Estimator
from models.capacity import Capacity
from models.node import Node
from models.pod import Pod


class FirstFitEstimator(Estimator):
    def estimate(
        self,
        initial_nodes: List[Node],
        node_capacity: Capacity,
        pods: Sequence[Pod],
        vector_converter: VectorConverter
    ) -> EstimationResult:
        nodes = initial_nodes
        nodes_created = 0

        for pod in pods:
            existing_node_used = False

            for node in nodes:
                if node.can_fit(pod):
                    existing_node_used = True
                    node.add_pod(pod)

            if not existing_node_used:
                new_node = Node.generate_with_capacity(node_capacity)
                new_node.add_pod(pod)
                nodes.append(new_node)

                nodes_created += 1
```

```
        return EstimationResult(nodes, nodes_created)

    def get_name(self):
        return "First Fit"


from typing import Sequence, List, Type

from estimator.converters.vector_converter import VectorConverter
from estimator.estimation_result import EstimationResult
from estimator.estimator import Estimator
from models.capacity import Capacity
from models.node import Node
from models.pod import Pod


class MyHybridWorstEstimator(Estimator):
    def estimate(
        self,
        initial_nodes: List[Node],
        node_capacity: Capacity,
        pods: Sequence[Pod],
        vector_converter: VectorConverter
    ) -> EstimationResult:
        nodes = initial_nodes
        nodes_created = 0

        def sort_key(px):
            scalar_px = vector_converter.convert(px.requests)
            return scalar_px

        pods = sorted(pods, key=lambda x: sort_key(x), reverse=True)

        for pod in pods:
            existing_node_used = False
            best_node_index = -1
            prev_node_index = -1
            max_score = 0

            for idx, node in enumerate(nodes):
                if not node.can_fit(pod):
                    continue

                scalar_node = vector_converter.convert(capacity=node.remaining_capacity)
                score = scalar_node
```

```
            if best_node_index == -1:
                max_score = score
                best_node_index = idx
                prev_node_index = idx
            else:
                if score > max_score:
                    prev_node_index = best_node_index
                    best_node_index = idx
                    max_score = score

        if best_node_index != -1:
            existing_node_used = True
            nodes[prev_node_index].add_pod(pod)

        if not existing_node_used:
            new_node = Node.generate_with_capacity(node_capacity)
            new_node.add_pod(pod)
            nodes.append(new_node)

            nodes_created += 1

        return EstimationResult(nodes, nodes_created)

    def get_name(self):
        return "Hybrid Worst Fit"


class MyHybridBestEstimator(Estimator):
    def estimate(
        self,
        initial_nodes: List[Node],
        node_capacity: Capacity,
        pods: Sequence[Pod],
        vector_converter: VectorConverter
    ) -> EstimationResult:
        nodes = initial_nodes
        nodes_created = 0

        for pod in pods:
            existing_node_used = False
            best_node_index = -1
            prev_node_index = -1
            min_score = 0
```

```
            for idx, node in enumerate(nodes):
                if not node.can_fit(pod):
                    continue

                scalar_node = vector_converter.convert(capacity=node.remaining_capacity)
                score = scalar_node

                if best_node_index == -1:
                    min_score = score
                    best_node_index = idx
                    prev_node_index = idx
                else:
                    if score < min_score:
                        prev_node_index = best_node_index
                        best_node_index = idx
                        min_score = score

            if best_node_index != -1:
                existing_node_used = True
                nodes[prev_node_index].add_pod(pod)

            if not existing_node_used:
                new_node = Node.generate_with_capacity(node_capacity)
                new_node.add_pod(pod)
                nodes.append(new_node)

                nodes_created += 1

        return EstimationResult(nodes, nodes_created)

    def get_name(self):
        return "Hybrid Best Fit"

from typing import Sequence, List, Type

from estimator.converters.vector_converter import VectorConverter
from estimator.estimation_result import EstimationResult
from estimator.estimator import Estimator
from models.capacity import Capacity
from models.node import Node
from models.pod import Pod


class NextFitEstimator(Estimator):
    def estimate(
```

```
        self,
        initial_nodes: List[Node],
        node_capacity: Capacity,
        pods: Sequence[Pod],
        vector_converter: VectorConverter
    ) -> EstimationResult:
        nodes = initial_nodes
        nodes_created = 0
        current_node = None

        for pod in pods:
            can_fit = False

            if current_node and current_node.can_fit(pod):
                can_fit = True
                current_node.add_pod(pod)

            if not can_fit:
                new_node = Node.generate_with_capacity(node_capacity)
                new_node.add_pod(pod)
                nodes.append(new_node)

                current_node = new_node
                nodes_created += 1

        return EstimationResult(nodes, nodes_created)

    def get_name(self):
        return "Next Fit"


class NextFitDecreasingEstimator(NextFitEstimator):
    def estimate(
        self,
        initial_nodes: List[Node],
        node_capacity: Capacity,
        pods: Sequence[Pod],
        vector_converter: VectorConverter
    ) -> EstimationResult:
        def sort_key(px):
            scalar_px = vector_converter.convert(px.requests)
            return scalar_px

        pods = sorted(pods, key=lambda x: sort_key(x), reverse=True)
        return super().estimate(
```

```
            initial_nodes=initial_nodes,
            node_capacity=node_capacity,
            pods=pods,
            vector_converter=vector_converter
        )

    def get_name(self):
        return "Next Fit Decreasing"


from typing import Sequence, List, Type

from estimator.converters.vector_converter import VectorConverter
from estimator.estimation_result import EstimationResult
from estimator.estimator import Estimator
from models.capacity import Capacity
from models.node import Node
from models.pod import Pod


class WorstFitEstimator(Estimator):
    def estimate(
        self,
        initial_nodes: List[Node],
        node_capacity: Capacity,
        pods: Sequence[Pod],
        vector_converter: VectorConverter
    ) -> EstimationResult:
        nodes = initial_nodes
        nodes_created = 0

        for pod in pods:
            existing_node_used = False
            best_node_index = -1
            max_score = 0

            for idx, node in enumerate(nodes):
                if not node.can_fit(pod):
                    continue

                scalar_node = vector_converter.convert(capacity=node.remaining_capacity)
                score = scalar_node

                if best_node_index == -1:
                    max_score = score
```

```
                best_node_index = idx
            else:
                if score > max_score:
                    best_node_index = idx
                    max_score = score

        if best_node_index != -1:
            existing_node_used = True
            nodes[best_node_index].add_pod(pod)

        if not existing_node_used:
            new_node = Node.generate_with_capacity(node_capacity)
            new_node.add_pod(pod)
            nodes.append(new_node)

            nodes_created += 1

        return EstimationResult(nodes, nodes_created)

    def get_name(self):
        return "Worst Fit"


from abc import ABCMeta, abstractmethod

from models.capacity import Capacity


class VectorConverter(metaclass=ABCMeta):
    @abstractmethod
    def convert(
        self,
        capacity: Capacity,
        a_1=1,
        a_2=1
    ) -> float:
        pass


from estimator.converters.vector_converter import VectorConverter
from models.capacity import Capacity


class SumVectorConverter(VectorConverter):
    def convert(
        self,
```

```
        capacity: Capacity,
        a_1=1,
        a_2=1
    ) -> float:
        return a_1 * capacity.cpu + a_2 * capacity.ram


from estimator.converters.vector_converter import VectorConverter
from models.capacity import Capacity


class MulVectorConverter(VectorConverter):
    def convert(
        self,
        capacity: Capacity,
        a_1=1,
        a_2=1
    ) -> float:
        return a_1 * capacity.cpu * a_2 * capacity.ram


class Capacity:
    def __init__(self, cpu, ram):
        self.cpu = cpu
        self.ram = ram

    def validate_self(self):
        if self.cpu < 0 or self.ram < 0:
            raise ValueError("Ресурсна ємність не може бути від'ємною")

    def __str__(self):
        return f"Capacity: cpu[{self.cpu}], ram[{self.ram}]"


import uuid
from typing import Sequence

from models.capacity import Capacity
from models.pod import Pod


class Node:
    def __init__(
        self,
        name: str,
        labels: Sequence[str],
```

```
        capacity: Capacity,
        **kwargs
    ):
        self.name = name
        self.labels = labels
        self.capacity = capacity
        self.kwargs = kwargs
        self.remaining_capacity = Capacity(capacity.cpu, capacity.ram)
        self.pods = []

    @staticmethod
    def generate_with_capacity(capacity: Capacity):
        return Node(
            name="Node_" + str(uuid.uuid4()),
            labels=[],
            capacity=capacity
        )

    def add_pod(self, pod: Pod):
        self.remaining_capacity.cpu -= pod.requests.cpu
        self.remaining_capacity.ram -= pod.requests.ram
        self.remaining_capacity.validate_self()

        self.pods.append(pod)

    def remove_pod(self, pod):
        self.remaining_capacity.cpu += pod.requests.cpu
        self.remaining_capacity.ram += pod.requests.ram

        self.pods.remove(pod)

    def can_fit(self, pod):
        return self.remaining_capacity.cpu >= pod.requests.cpu and \
               self.remaining_capacity.ram >= pod.requests.ram

    def __repr__(self):
        return f"Node `{self.name}`: Max: {self.capacity}, Available
{self.remaining_capacity}"


from models.capacity import Capacity


class Pod:
    def __init__(
```

```
        self,
        name: str,
        requests: Capacity,
    ):
        self.name = name
        self.requests = requests

    def __str__(self):
        return f"Node `{self.name}`: Requests: {self.requests}"


import logging
import random
import uuid

from models.capacity import Capacity
from models.pod import Pod


def generate_pods_random_sample(
    pods_count,
    cpu_min,
    cpu_max,
    ram_min,
    ram_max
):
    generated_pods = []

    for i in range(pods_count):
        current_cpu = random.randint(cpu_min, cpu_max)
        current_ram = random.randint(ram_min, ram_max)
        current_name = "Pod_" + str(uuid.uuid4())
        current_capacity = Capacity(
            current_cpu,
            current_ram
        )

        new_pod = Pod(
            name=current_name,
            requests=current_capacity
        )

        logging.info(f"Generated Pod: {new_pod}")
        generated_pods.append(new_pod)
```

```
    return generated_pods


import math
from statistics import mean

from estimator.converters.mul_vector_converter import MulVectorConverter
from estimator.estimators.best_fit_estimator import BestFitEstimator
from estimator.estimators.estimator_decorator import SORT_INCREASING,
SORT_DECREASING, AS_IS, \
    EstimatorDecorator
from estimator.estimators.first_fit_estimator import FirstFitEstimator
from estimator.estimators.my_hybrid_estimator import MyHybridWorstEstimator, \
    MyHybridBestEstimator
from estimator.estimators.next_fit_estimator import NextFitEstimator
from estimator.estimators.worst_fit_estimator import WorstFitEstimator
from models.capacity import Capacity
from simulator.simulate import generate_pods_random_sample

if __name__ == "__main__":
    generated_pods = generate_pods_random_sample(
        pods_count=3000,
        cpu_min=100,
        cpu_max=350,
        ram_min=256,
        ram_max=712,
    )

    target_node_capacity = Capacity(
        cpu=1000,
        ram=2048
    )

    estimators = [
        NextFitEstimator(),
        FirstFitEstimator(),
        BestFitEstimator(),
        WorstFitEstimator(),
        MyHybridWorstEstimator(),
        MyHybridBestEstimator(),
    ]

    sum_cpu = sum([x.requests.cpu for x in generated_pods])
    sum_ram = sum([x.requests.ram for x in generated_pods])    lower_bound = max(
```

```
math.ceil(sum_cpu / target_node_capacity.cpu),
        math.ceil(sum_ram / target_node_capacity.ram),
    )

    for estimator in estimators:
        for order in (SORT_DECREASING, SORT_INCREASING, AS_IS):
            current_estimator = EstimatorDecorator(
                estimator=estimator,
                name_suffix=order,
                sort_order=order
            )

            result = current_estimator.estimate(
                initial_nodes=[],
                node_capacity=target_node_capacity,
                pods=generated_pods,
                vector_converter=MulVectorConverter()
            )

            name = ''.join([x[0] for x in current_estimator.get_name().split()])
            used = result.new_nodes_created
            cpu_utilization = round(mean([1 - (x.remaining_capacity.cpu / x.capacity.cpu)
for x in result.nodes]) * 100, 2)
            ram_utilization = round(mean([1 - (x.remaining_capacity.ram / x.capacity.ram)
for x in result.nodes]) * 100, 2)

            print(f"{name},{used},{cpu_utilization},{ram_utilization}")
    print(f"LB: {lower_bound}")


FROM python:3.8

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD ["gunicorn", "app:app", "--log-level=INFO",
     "--threads=1", "--workers=1", "-b", "0.0.0.0:8000
```